

MANNING



Redis

实战

Redis
IN ACTION

〔美〕 Josiah L. Carlson 著
黄健宏 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

□ □

[□□□□](#)

[□□□□](#)

[□□□□](#)

[□□](#)

[□□□](#)

[□□□□□□□□□□□□](#)

[□□□□](#)

[□□□□](#)

[□](#)

[□□](#)

[□□](#)

[□□□□](#)

[□□□□](#)

[□□□□□□□](#)

[□□□□□□](#)

[□□□□](#)

[□□□□□□](#)

[□□□□_□□](#)

[□1□_□□Redis](#)

1.1 Redis

1.1.1 Redis

1.1.2

1.1.3 Redis

1.2 Redis

1.2.1 Redis

1.2.2 Redis

1.2.3 Redis

1.2.4 Redis

1.2.5 Redis

1.3 Redis

1.3.1

1.3.2

1.3.3

1.4

1.5

2 Redis Web

2.1 cookie

2.2 Redis

2.3

2.4

[2.5 環境](#)

[2.6 例](#)

[環境構築](#)

[3 Redis](#)

[3.1 環境](#)

[3.2 例](#)

[3.3 例](#)

[3.4 例](#)

[3.5 環境](#)

[3.6 環境](#)

[3.7 環境](#)

[3.7.1 例](#)

[3.7.2 Redis](#)

[3.7.3 環境](#)

[3.8 例](#)

[4 環境構築](#)

[4.1 環境](#)

[4.1.1 環境](#)

[4.1.2 AOF](#)

[4.1.3 例/AOF](#)

[4.2 例](#)

[4.2.1 Redisのインストール](#)

[4.2.2 Redisの起動](#)

[4.2.3 設定](#)

[4.2.4 接続](#)

[4.3 高可用性](#)

[4.3.1 RedisのAOF](#)

[4.3.2 RedisのReplication](#)

[4.4 Redisのセキュリティ](#)

[4.4.1 Redisのパスワード](#)

[4.4.2 RedisのACL](#)

[4.4.3 RedisのSSL/TLS](#)

[4.5 Redisのバックアップと復元](#)

[4.6 Redisの監視とログ](#)

[4.7 結論](#)

[5 Redisの応用](#)

[5.1 Redisの応用1](#)

[5.1.1 応用1.1](#)

[5.1.2 応用1.2](#)

[5.2 Redisの応用2](#)

[5.2.1 応用2.1 Redisの応用](#)

[5.2.2 応用2.2 Redisの応用](#)

[5.2.3 安装Redis](#)

[5.3 配置Redis](#)

[5.3.1 配置Redis](#)

[5.3.2 配置Redis](#)

[5.4 使用Redis](#)

[5.4.1 使用Redis](#)

[5.4.2 使用Redis](#)

[5.4.3 使用Redis](#)

[5.5 总结](#)

[6 Redis进阶](#)

[6.1 进阶](#)

[6.1.1 进阶](#)

[6.1.2 进阶](#)

[6.2 进阶](#)

[6.2.1 进阶](#)

[6.2.2 进阶](#)

[6.2.3 使用Redis](#)

[6.2.4 进阶](#)

[6.2.5 进阶](#)

[6.3 进阶](#)

[6.3.1 进阶](#)

[6.3.2 配置](#)

[6.3.3 配置](#)

[6.3.4 配置](#)

[6.4 配置](#)

[6.4.1 配置](#)

[6.4.2 配置](#)

[6.5 配置](#)

[6.5.1 配置](#)

[6.5.2 配置](#)

[6.6 Redis配置](#)

[6.6.1 配置](#)

[6.6.2 配置](#)

[6.6.3 配置](#)

[6.6.4 配置](#)

[6.7 配置](#)

[7 Redis配置](#)

[7.1 Redis配置](#)

[7.1.1 配置](#)

[7.1.2 配置](#)

[7.2 配置](#)

[7.2.1 配置](#)

7.2.2 関数型プログラミング

7.3 関数

7.3.1 関数の定義

7.3.2 関数の呼び出し

7.3.3 関数の戻り値

7.3.4 関数の副作用

7.4 関数

7.4.1 関数の定義

7.4.2 関数の呼び出し

7.5 関数

8 関数型プログラミング

8.1 関数

8.1.1 関数

8.1.2 関数

8.2 関数

8.3 関数型プログラミング

8.4 関数型プログラミング

8.5 API

8.5.1 APIの定義

8.5.2 関数

8.5.3 関数型プログラミング

8.6 小结

数据库系统组成

9 数据库系统

9.1 数据库

9.1.1 数据库的概念

9.1.2 数据库系统的组成

9.1.3 数据库系统的层次结构

9.2 数据库设计

9.2.1 数据库设计的内容

9.2.2 数据库设计的过程

9.3 数据库的维护

9.3.1 数据库的备份与恢复

9.3.2 数据库的优化

9.3.3 数据库的安全性

9.4 小结

10 数据库Redis

10.1 数据库Redis

10.2 数据库Redis的安装

10.2.1 数据库Redis的安装

10.2.2 数据库Redis的配置

10.3 数据库Redis的测试

[10.3.1 10.3.1](#)

[10.3.2 10.3.2](#)

[10.3.3 10.3.3](#)

[10.4 10.4](#)

[11 Redis Lua 11](#)

[11.1 C 11.1](#)

[11.1.1 Lua Redis 11.1.1](#)

[11.1.2 11.1.2](#)

[11.2 Lua 11.2](#)

[11.2.1 Lua 11.2.1](#)

[11.2.2 11.2.2](#)

[11.2.3 Lua 11.2.3](#)

[11.3 WATCH/MULTI/EXEC 11.3](#)

[11.3.1 11.3.1](#)

[11.3.2 11.3.2](#)

[11.4 Lua 11.4](#)

[11.4.1 11.4.1](#)

[11.4.2 11.4.2](#)

[11.4.3 11.4.3](#)

[11.4.4 11.4.4](#)

[11.5 11.5](#)

[第A章 Redis入门](#)

[A.1 Debian Linux/Ubuntu Linux安装Redis](#)

[A.2 OS X安装Redis](#)

[A.3 Windows安装Redis](#)

[A.3.1 Windows安装Redis](#)

[A.3.2 Windows Redis](#)

[A.3.3 Windows Python](#)

[A.4 Redis](#)

[第B章 Redis进阶](#)

[B.1 Redis](#)

[B.2 Redis](#)

[B.3 Redis](#)

[B.4 Redis](#)

[B.5 Redis](#)

[B.6 Redis](#)

[附录](#)

本站所有资源均来自网络，如有侵权，请联系删除。
ePUBw.COM 本站所有资源均来自网络，如有侵权，请联系删除。

本站所有资源均来自网络，如有侵权，请联系删除。

□ □ □ □

Redis

ISBN 978-7-115-40284-4

□ □

[illegible][illegible][illegible]

11

- ☐ [] Josiah L. Carlson

1 1 1




- □□□□□□□□ □□□□□□□□11□

□□ 100164 □□□□ 315@ptpress.com.cn

□□□□

□□□□□□□□Redis5□□□□□□□□□□□□□□Redis□□□□□□
□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□Redis□□□□□□□□
□□

□□□□□□□□□□□□□□Redis□□□□□□□□Redis□□□□□□□□□□
5□□□□□□□□□5□□□□□□□□□□□□□□Redis□□□□□□□□□□cookie□
□□□□□□□□□□□□□□□□□□□□□□□□Redis□□□□□□□□□□□□□□□□
□Redis□□□□□□□□□□□□□□□□□□□□□□□□Redis□□□□□□□□□□
□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□□Redis□□□□□□□□□□
Redis□□□□□□□□□□Lua□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Redis□□
□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□NoSQL□□□□□□□□□□Redis□□□□□

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□□□

□□

□□□□□□□□□□ See Iuan□□□□□□□□ Mikela□

□□□□ ePUBw.COM □□□□ ePUBw.COM □□□□

□□□□□□□□□□□□□□

████████████████████

████████████████████redisinaction.com████████████████████

██

██

██

████████ePUBw.COM████████ePUBw.COM████████

████████████████████████████████████

📄📄📄

👤huangz👤1990 📄📄📄📄📄📄📄📄📄📄📄Redis
📄📄📄📄📄Redis📄📄📄Disque 📄📄📄📄📄📄📄📄📄📄📄
📄📄📄📄📄📄huangz.me📄

📄📄📄ePUBw.COM📄📄📄ePUBw.COM 📄📄📄

📄📄📄📄📄📄📄📄📄

□

Redis 是 3 个数据库的集合，它支持字符串、哈希、列表、集合、有序集合、位图、超文本等数据类型。它支持 on-disk SQL database 模式，支持主从复制、集群、持久化、事务、Lua 脚本等特性。

Redis 支持 JavaScript 的 tracker 模式，支持 page view 跟踪，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。

Redis 支持 schema 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 prototype 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 pop 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 C 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。

Redis 支持 Redis 2.6 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 Redis 2.6 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 Redis 2.6 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 Redis 2.6 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。Redis 支持 Redis 2.6 模式，支持 Redis 的持久化、主从复制、集群、事务、Lua 脚本等特性。

Redis BSD
Redis Redis Redis

Redis —
Redis Redis *Redis in Action* API
Redis

[class="sigil_not_in_toc"] { color:#880000; }
Redis Redis Josiah
Redis — Redis Group

Redis Redis
Redis Redis Redis

— Salvatore Sanfilippo “Redis”

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

□□

Chris Testa 在 Google 工作，2010 年 3 月
Chris 在 Google 工作，2010 年 3 月
Chris 在 Google 工作，2010 年 3 月

Chris 在 Redis 工作，2010 年 3 月
Chris 在 Redis 工作，2010 年 3 月
Chris 在 Redis 工作，2010 年 3 月

Redis 在 Twitter 工作，2010 年 3 月
Redis 在 Twitter 工作，2010 年 3 月
Redis 在 Twitter 工作，2010 年 3 月

2011 年 9 月，Manning 出版 Michael
Stephens 的《Redis 入门》，Michael
Stephens 在 10 月 2 日出版 Michael

Michael 在 Manning 出版 Redis
Michael 在 Manning 出版 Redis
Michael 在 Manning 出版 Redis

Redis
Manning
Michael 17 Redis
——
20

Redis

ePUBw.COM ePUBw.COM

□□

□□□□□□□□Manning□Beth Lexleigh□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□

□□□□□□□□□Bert Bates□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□

□□□Salvatore Sanfilippo□□□□□□□Redis□□□□□□□□□□□□□□
□□□□□

□□□Pieter Noordhuis□□□□□□□Redis□□□□□□□□□□□□
RedisConf 2012□□□□□□□□□□□□□□□Redis□□□□□□□□□□□□
□□□□□□□□□□□□□□□Redis□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□James Phillips□Kevin Chang□
Nicholas Lindgren□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□Eric Van Dewoestine□□□□□□□□□□□□□□□Java□□
□□□□□□□□□□□□□□□□GitHub□□□□□[https://github.com/
josiahcarlson/redis-in-action](https://github.com/josiahcarlson/redis-in-action)□

□□□□Amit Nandi□Bennett Andrews□Bobby Abraham□
Brian Forester□Brian Gyss□Brian McNamara□Daniel

Sundman□David Miller□Felipe Gutierrez□Filippo Pacini□
Gerard O’Sullivan□JC Pretorius□Jonathan Crawley□Joshua
White□Leo Cassarani□Mark Wigmans□Richard Clayton□Scott
Lyons□Thomas O’Rourke□Todd Fiala□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□Manning□□Redis□□□□□□□□□□□□□□□□□□□□□□
□□□□□

□□□□□□□□See Luan□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□□□□□

□□□□

□□□□Redis□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□
□□Salvatore Sanfilippo□□□□□□□□□□□□□□□□□□□□Redis□□
□□□□□□□□□□□□□□□□Python□□□□□□Redis□□□□□□□□□□□□
Python□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□Python□□□□□□□□□□Python 2.7.x□□□□Python□□□□
□Python language tutorial□□□□□□□□□□Python□□□□□□□□□□□□
□□□□□□□□□□□□□□□□Python□□□□□□□□□□□□Java□□□□JavaScript□
□□□□Ruby□□□□□□□□□□□□□□□□□□□□□□□□Python□□□□□□□□□□
□□□□□□□□Python□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□Redis□□□□□□□□□□□□□□□□1□□□2□□□□□□□□□□□□
□□□□□□Redis□□□□□□Python□□□□□□□□□□□□□□□□□□□□1□□□2□□□
□□□□□□1□□□2□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□□□□□3□
□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□□□□□4□□□□Redis□□□□□□□□
□□□□□□□□□□□□

□□□□□□□□Redis□□□□□□□□□□□□□□□□1□□□3□——□□□□□□□□□□□□
□□□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□□□□□Redis□□□□

前言

本书是第3版，它涵盖了Redis的所有功能。Redis是一个开源的、基于内存的、键值对数据库。它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、位图等。Redis还支持Lua脚本，可以实现复杂的业务逻辑。本书详细介绍了Redis的安装、配置、使用、性能优化、高可用、持久化、备份与恢复、安全等方面的知识。本书适合Redis的初学者和进阶者阅读。

第1章 Redis的安装与配置。Redis的安装与配置是本书的第一章，介绍了Redis的下载、编译、安装、配置、启动、停止、重启、升级、备份与恢复、安全等方面的知识。

第2章 Redis的数据类型与操作。Redis的数据类型与操作是本书的第二章，介绍了Redis支持的多种数据类型，包括字符串、列表、集合、有序集合、哈希表、位图等，以及对这些数据类型的操作。

第3章 Redis的持久化与备份。Redis的持久化与备份是本书的第三章，介绍了Redis的持久化机制，包括RDB和AOF，以及Redis的备份与恢复。

第4章 Redis的高可用与性能优化。Redis的高可用与性能优化是本书的第四章，介绍了Redis的高可用架构，包括主从复制、哨兵、集群，以及Redis的性能优化。

Redis 4.4和4.5版本。Redis 4.4和4.5版本是Redis的4.x版本，它们支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、位图等。Redis 4.4和4.5版本还支持Lua脚本，可以实现复杂的业务逻辑。

5 Redis IP

6

7

8 Twitter API

9 Redis

10 Redis

11 Lua Redis
 Lua

A Linux OS X Windows 3 Redis
 Python Python Redis

B Redis Python
 Redis Redis

www.ePUBw.COM www.ePUBw.COM www.ePUBw.COM

www.ePUBw.COM

□□□□

□□□□□□□□ Josiah Carlson □□□□□□□□□□□□□□□□□□□□□□□□
□□ Josiah □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Josiah □□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□ Networks in Motion □□□□□□□□□□
□□□□□□ Networks in Motion □□□□□□□□ Josiah □□□□□□ GPS □□□□□□□□
□□□□□□□□□□

□□□ Networks in Motion □□□□□ Josiah □□□ Google □□□□□□□□□□
□□ Adly □□
Josiah □□□ Redis □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□ ChowNow □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□ Josiah □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Redis □
□□□

□□□□ ePUBw.COM □□□□ ePUBw.COM □□□□

□□□□□□□□□□□□□□□□□□□□

□□□□ □□

□□□□□□□□Redis□□□□□□□Redis□□□□□□□□□□□□□□□□
□□□Redis□□□□□□□□□□□□□□□□

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□□□

1 Redis

Redis

- Redis 是什么
- Redis 的部署
- 使用 Python 操作 Redis
- Redis 的持久化

Redis 是一个开源的、高性能的、分布式的键值数据库。

Redis 支持多种数据类型，如字符串、列表、集合、有序集合、哈希等。

Redis 支持主从复制、哨兵、集群等高级功能。

Redis 支持持久化，可以将数据保存到磁盘，防止数据丢失。

Redis 支持客户端分片，可以支持更多的客户端连接。

Redis 的内存使用非常灵活，可以配置内存上限。

Redis 的部署非常简单，可以在单机上部署，也可以在集群中部署。

Redis 的社区非常活跃，有很多第三方库和工具。

Redis 的文档非常详细，可以帮助你快速上手。

Redis 的官方网站是 <https://redis.io/>。

Redis 的 GitHub 仓库是 <https://github.com/redis/redis>。

Redis Redis

Redis

Redis
Redis Redis

Redis Python A Redis Python

Redis Python
 Ruby Java JavaScript Node.js
<https://github.com/josiahcarlson/redis-in-action> Spring
<http://www.springsource.org/spring-data/redis>
 Spring Redis

□□□□ePUBw.COM□□□□ePUBw.COM□□□□

[illegible]

1.1 Redis

Redis 是一个开源的 Redis 数据库。Redis 是一个非关系型数据库（non-relational database），它使用 key-value 的映射（mapping）来存储数据。key 的长度在 1-512 字节之间，value 的长度在 1-512 字节之间。Redis 是一个单线程的数据库，它使用单线程来保证数据的一致性。Redis 是一个分布式数据库，它支持主从复制和集群模式。

1.1.1 Redis 数据库

Redis 是一个开源的数据库，它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、超文本标记语言（HTML）文档、位图、地理索引等。Redis 是一个高性能的数据库，它支持每秒处理 100,000 个请求。Redis 是一个分布式数据库，它支持主从复制和集群模式。

Redis 是一个开源的数据库，它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、超文本标记语言（HTML）文档、位图、地理索引等。Redis 是一个高性能的数据库，它支持每秒处理 100,000 个请求。Redis 是一个分布式数据库，它支持主从复制和集群模式。Redis 是一个开源的数据库，它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、超文本标记语言（HTML）文档、位图、地理索引等。Redis 是一个高性能的数据库，它支持每秒处理 100,000 个请求。Redis 是一个分布式数据库，它支持主从复制和集群模式。

Redis 是一个开源的数据库，它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、超文本标记语言（HTML）文档、位图、地理索引等。Redis 是一个高性能的数据库，它支持每秒处理 100,000 个请求。Redis 是一个分布式数据库，它支持主从复制和集群模式。Redis 是一个开源的数据库，它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、超文本标记语言（HTML）文档、位图、地理索引等。Redis 是一个高性能的数据库，它支持每秒处理 100,000 个请求。Redis 是一个分布式数据库，它支持主从复制和集群模式。

1. Redis 是什么？Redis 是一个开源的、基于内存的、支持多种数据类型的数据库。它支持字符串、哈希、列表、集合、有序集合、位图、超文本标记语言（HTML）等数据类型。Redis 还支持主从复制、持久化、事务、Lua 脚本等功能。

1-1 Redis 的部署和配置

1-1 Redis 的部署和配置

名称	语言	数据类型	特性	其他
Redis	C语言 in-memory 持久化	字符串 哈希 列表 集合 有序集合 位图 超文本标记语言（HTML）	主从复制 持久化 事务 Lua 脚本 批量操作 部分操作	主从复制 master/slave replication 持久化 stored procedure
memcached	C语言 内存	字符串 哈希 列表 集合 有序集合 位图 超文本标记语言（HTML）	主从复制 持久化 事务 Lua 脚本 批量操作 部分操作	主从复制 master/slave replication 持久化 stored procedure

DB	Category	Features	Operations	Capabilities
MySQL	Relational	<ul style="list-style-type: none"> ACID-compliant Supports InnoDB engine Supports views Supports spatial data 	SELECT INSERT UPDATE DELETE	ACID-compliant InnoDB engine Supports master/master replication
PostgreSQL	Relational	<ul style="list-style-type: none"> ACID-compliant Supports multi-master replication 	SELECT INSERT UPDATE DELETE	ACID-compliant Supports multi-master replication
MongoDB	NoSQL	<ul style="list-style-type: none"> Schema-less Supports BSON 	SELECT INSERT UPDATE DELETE	map-reduce spatial index

1.1.2 Redis

Redis is a key-value database. It is a “distributed database” Redis is a key-value database. It is a “distributed database”

point-in-time dump“
dump-to-disk
append-only
sync4
[]

Redis Redis Redis
Redis Redis failover
Redis
copy
Redis

1.1.3 Redis

memcached APPEND
memcached APPEND
memcached blacklist
memcached Redis
LIST SET

```

Redis memcached

```

1. aggregates
 2. random read
 3. random write
 4. Redis
 5. atomic INCR
 6. Redis
 7. parser
 8. optimizer
 9. Redis

Redis 数据库的持久化策略

Redis 数据库的持久化策略主要有两种：RDB 持久化和 AOF 持久化。

RDB 持久化（Redis Database Backup）：通过定期将内存中的数据进行快照（Snapshot）保存到磁盘。快照文件通常命名为 dump.rdb。

AOF 持久化（Append Only File）：通过记录每一个写操作（Write Command）到磁盘。AOF 文件通常命名为 appendonly.log。

6 task queue distribute utility “Web web-sacle technology”

Redis
Redis
Redis

ePUBw.COM ePUBw.COM

1.2 Redis

```

1-1 Redis 5
STRING LIST SET HASH ZSET
Redis 5 DEL TYPE RENAME
Redis 3 Redis

```

Redis STRING LIST HASH 3
semantics
Redis SET ZSET Redis
1-2 Redis 5
Redis

1-2 Redis5

名前	データ型	初期値
STRING	文字列	空文字列 increment decrement

Redis	Python	Python
LIST	Ordered list	Ordered list trim pop
SET	unordered collection	unordered collection
HASH	Hash	Hash
ZSET	member score	range

Redis 3 commands

<http://redis.io/commands>

Redis 5 Python A Redis Python Python Redis

Redis Python redis-py
Redis

Redis Python A
Redis Python A
Debian <http://redis.io/download>
Redis make && sudo make install
sudo python -m easy_install redis hiredis hiredis
C Redis

Python
Python Redis
Python

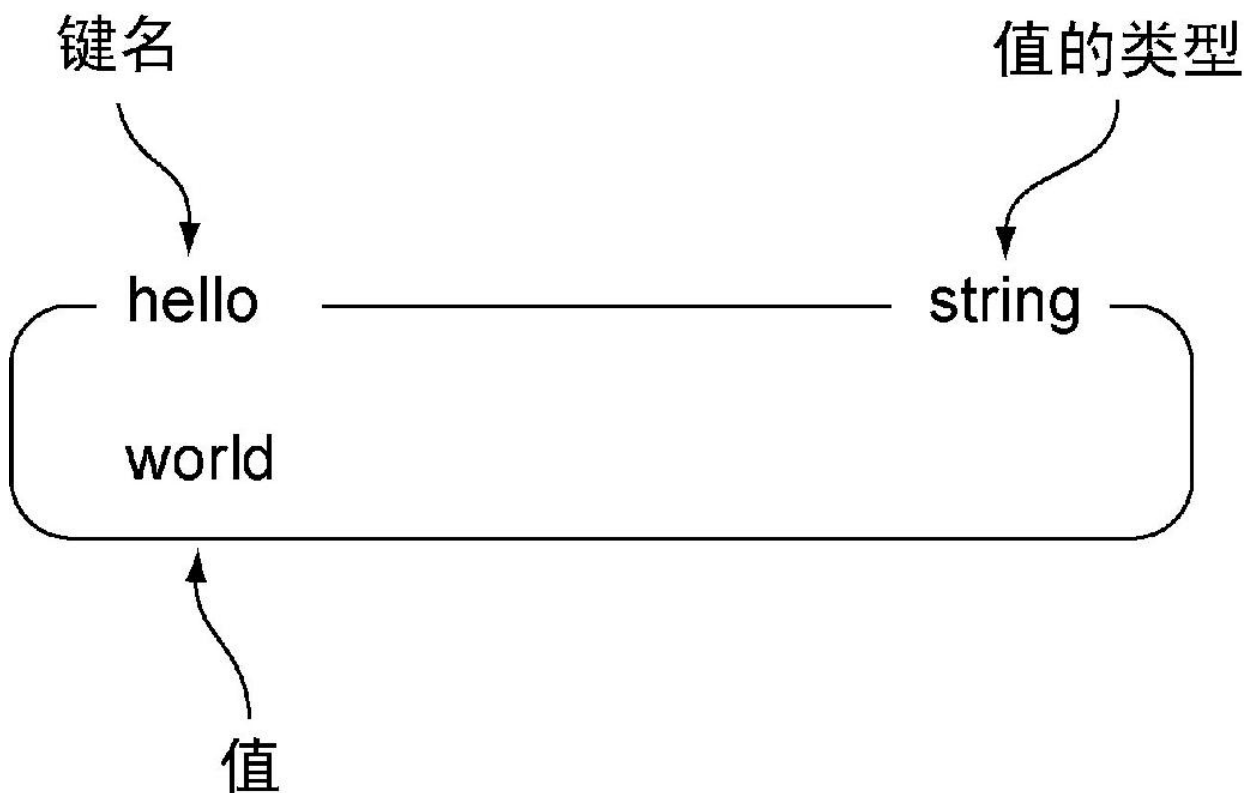
Python
Ruby Java JavaScript <https://github.com/josiahcarlson/redis-in-action> Python
Python

Python
Redis Redis Python
redis-cli Redis Redis
STRING

1.2.1 Redis数据类型

Redis数据类型分为字符串、哈希、列表、集合、有序集合、位图、超文本等。本章主要介绍字符串、哈希、列表、集合、有序集合、位图、超文本等数据类型。本章以字符串为例，介绍Redis数据类型的基本概念、操作命令、数据类型、数据类型转换、数据类型操作等。

key name 1-1
hello world



1-1 字符串数据类型hello world

Redis数据类型分为字符串、哈希、列表、集合、有序集合、位图、超文本等。本章主要介绍字符串、哈希、列表、集合、有序集合、位图、超文本等数据类型。本章以字符串为例，介绍Redis数据类型的基本概念、操作命令、数据类型、数据类型转换、数据类型操作等。

redis-cli SET GET DEL 1-3 3

1-3 字符串

命令	返回值
GET	字符串
SET	字符串
DEL	删除成功删除的值的数量

图例 1-1 SET、GET、DEL 命令

SET 命令在执行成功时返回 OK，Python 客户端会将这个 OK 转换成 True。

获取存储在键 hello 中的值。

在对值进行删除的时候，DEL 命令将返回被成功删除的值的数量。

```
$ redis-cli
redis 127.0.0.1:6379> set hello world
OK
redis 127.0.0.1:6379> get hello
"world"
redis 127.0.0.1:6379> del hello
(integer) 1
redis 127.0.0.1:6379> get hello
(nil)
redis 127.0.0.1:6379>
```

启动 redis-cli 客户端。

将键 hello 的值设置为 world。

键的值仍然是 world，跟我们刚才设置的一样。

删除这个键值对。

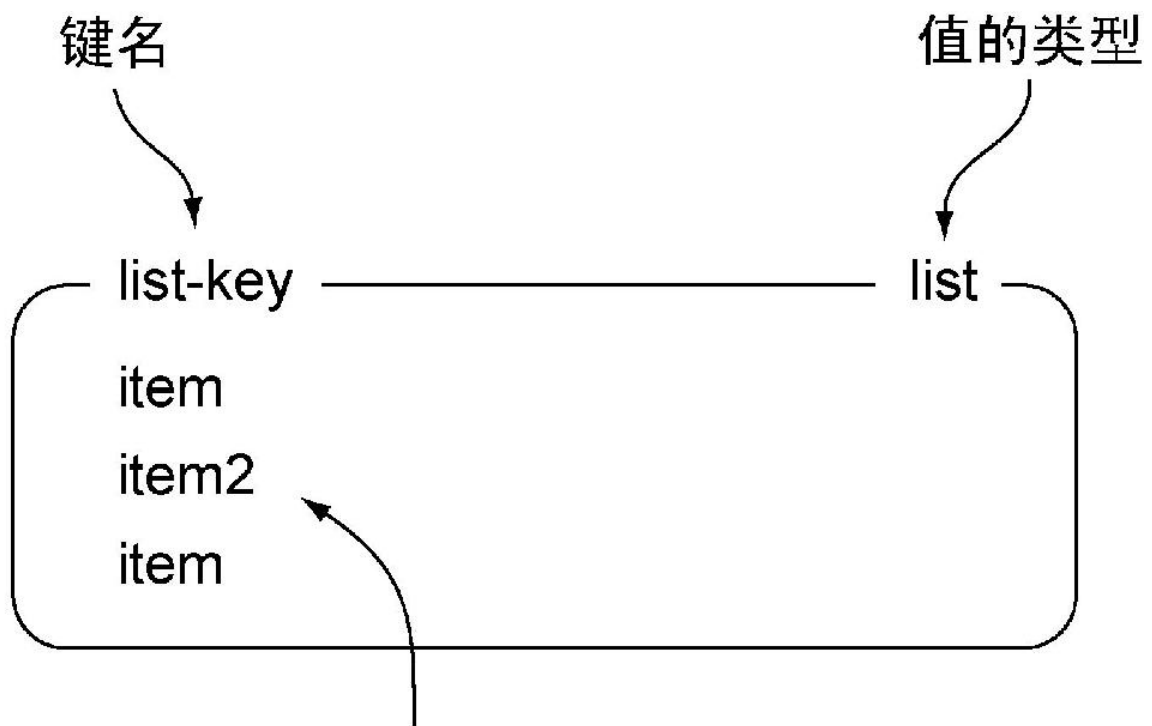
因为键的值已经不存在，所以尝试获取键的值将得到一个 nil，Python 客户端会将这个 nil 转换成 None。

在 **redis-cli** 中，Redis 命令的格式如下：

GET、SET、DEL 命令在 Redis 中都是原子操作，即在一个 Redis 实例中，同一时间只能有一个客户端执行这些命令。Redis 命令的格式如下：

1.2.2 Redis列表

Redis列表是linked-list数据结构实现的，列表中的元素都是字符串。列表中的元素可以重复出现，列表中的元素可以按1-2的顺序访问。



1-2 list-key 3

Redis列表的常用命令有LPUSH、RPUSH、LPOP、RPOP、LINDEX、LRANGE等。LPUSH和RPUSH分别向列表的左端和右端添加元素。LPOP和RPOP分别从列表的左端和右端移除元素。LINDEX和LRANGE分别用于获取列表中的元素。列表中的元素可以按1-2的顺序访问。

图1-4 列表

命令	功能
RPush	向列表右侧添加元素
LRange	返回列表中的元素范围
LIndex	返回列表中的元素
LPop	从列表左侧弹出元素

图1-2 RPush、LRange、LIndex、LPop命令

使用0为范围的起始索引，-1为范围的结束索引，可以取出列表包含的所有元素。

```
redis 127.0.0.1:6379> rpush list-key item
(integer) 1
redis 127.0.0.1:6379> rpush list-key item2
(integer) 2
redis 127.0.0.1:6379> rpush list-key item
(integer) 3
redis 127.0.0.1:6379> lrange list-key 0 -1
1) "item"
2) "item2"
3) "item"
redis 127.0.0.1:6379> lindex list-key 1
"item2"
redis 127.0.0.1:6379> lpop list-key
"item"
redis 127.0.0.1:6379> lrange list-key 0 -1
1) "item2"
2) "item"
redis 127.0.0.1:6379>
```

在向列表推入新元素之后，该命令会返回列表当前的长度。

在向列表推入新元素之后，该命令会返回列表当前的长度。

使用 LINDEX 可以从列表里面取出单个元素。

从列表里面弹出一个元素，被弹出的元素将不再存在于列表。

Redis 列表 Redis

——Redis

Redis 3 版本开始支持 Redis 数据类型

1.2.3 Redis

Redis 数据类型分为字符串、哈希、列表、集合、有序集合、位图、超文本标记语言（HTML）等。集合（Set）数据类型用于存储无序且不重复的元素。图 1-3 展示了 Redis 集合（Set）的数据结构。

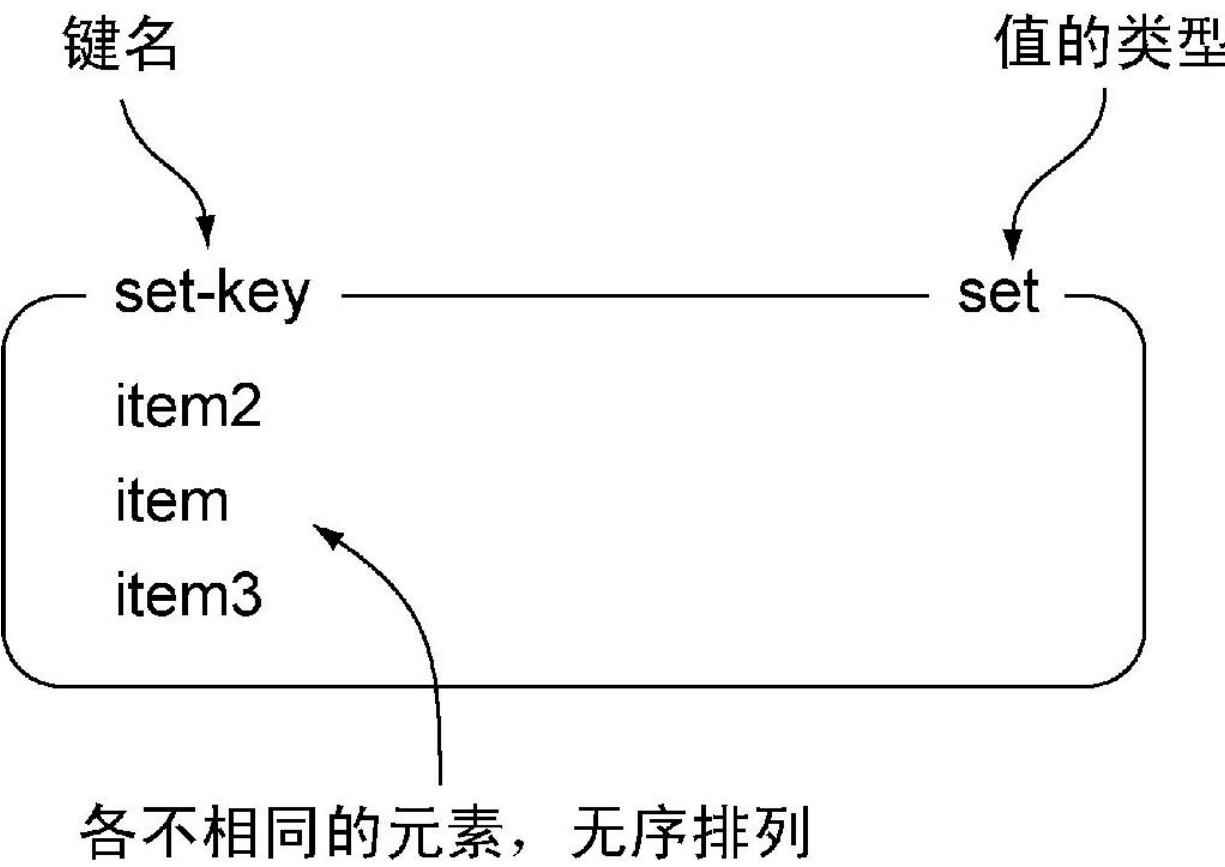


图 1-3 set-key 集合数据类型

Redis 的 unordered 集合类型是集合。SADD 命令用于向集合中添加元素。SREM 命令用于从集合中移除元素。SISMEMBER 命令用于检查元素是否存在于集合中。SMEMBERS 命令用于获取集合中的所有元素。1-3 表示集合中的元素。1-5 表示集合中的元素。

1-3 SADD SMEMBERS SISMEMBER SREM

```
redis 127.0.0.1:6379> sadd set-key item
(integer) 1
redis 127.0.0.1:6379> sadd set-key item2
(integer) 1
redis 127.0.0.1:6379> sadd set-key item3
(integer) 1
redis 127.0.0.1:6379> sadd set-key item
(integer) 0
redis 127.0.0.1:6379> smembers set-key
1) "item"
2) "item2"
3) "item3"
redis 127.0.0.1:6379> sismember set-key item4
(integer) 0
redis 127.0.0.1:6379> sismember set-key item
(integer) 1
redis 127.0.0.1:6379> srem set-key item2
(integer) 1
redis 127.0.0.1:6379> srem set-key item2
(integer) 0
redis 127.0.0.1:6379> smembers set-key
1) "item"
2) "item3"
redis 127.0.0.1:6379>
```

在尝试将一个元素添加到集合的时候，命令返回 1 表示这个元素被成功地添加到了集合里面，而返回 0 则表示这个元素已经存在于集合中。

获取集合包含的所有元素将得到一个由元素组成的序列，Python 客户端会将这个序列转换成 Python 集合。

检查一个元素是否存在于集合中，Python 客户端会返回一个布尔值来表示检查结果。

在使用命令移除集合中的元素时，命令会返回被移除元素的数量。

1-5

--	--

명령어	기능
SADD	집합에 원소 추가
SMEMBERS	집합의 모든 원소 반환
SISMEMBER	집합에 원소 존재 여부 확인
SREM	집합에서 원소 제거

집합 연산 명령어로는 SINTER, SUNION, SDIFF, 3개 집합의 교집합을 구하는 SINTER 3, 3개 집합의 차집합을 구하는 SDIFF 3, 7개 집합의 교집합을 구하는 SINTER 7 등이 있다. Redis는 5개 집합의 교집합을 구하는 SINTER 5 명령어를 제공한다.

1.2.4 Redis 설치

Redis는 Linux, macOS, Windows에서 실행 가능하다. 이 문서에서는 Linux에서 Redis를 설치하는 방법을 소개한다. Redis 1-4 버전은 64비트와 32비트 버전이 있지만, 이 문서에서는 64비트 버전을 소개한다.

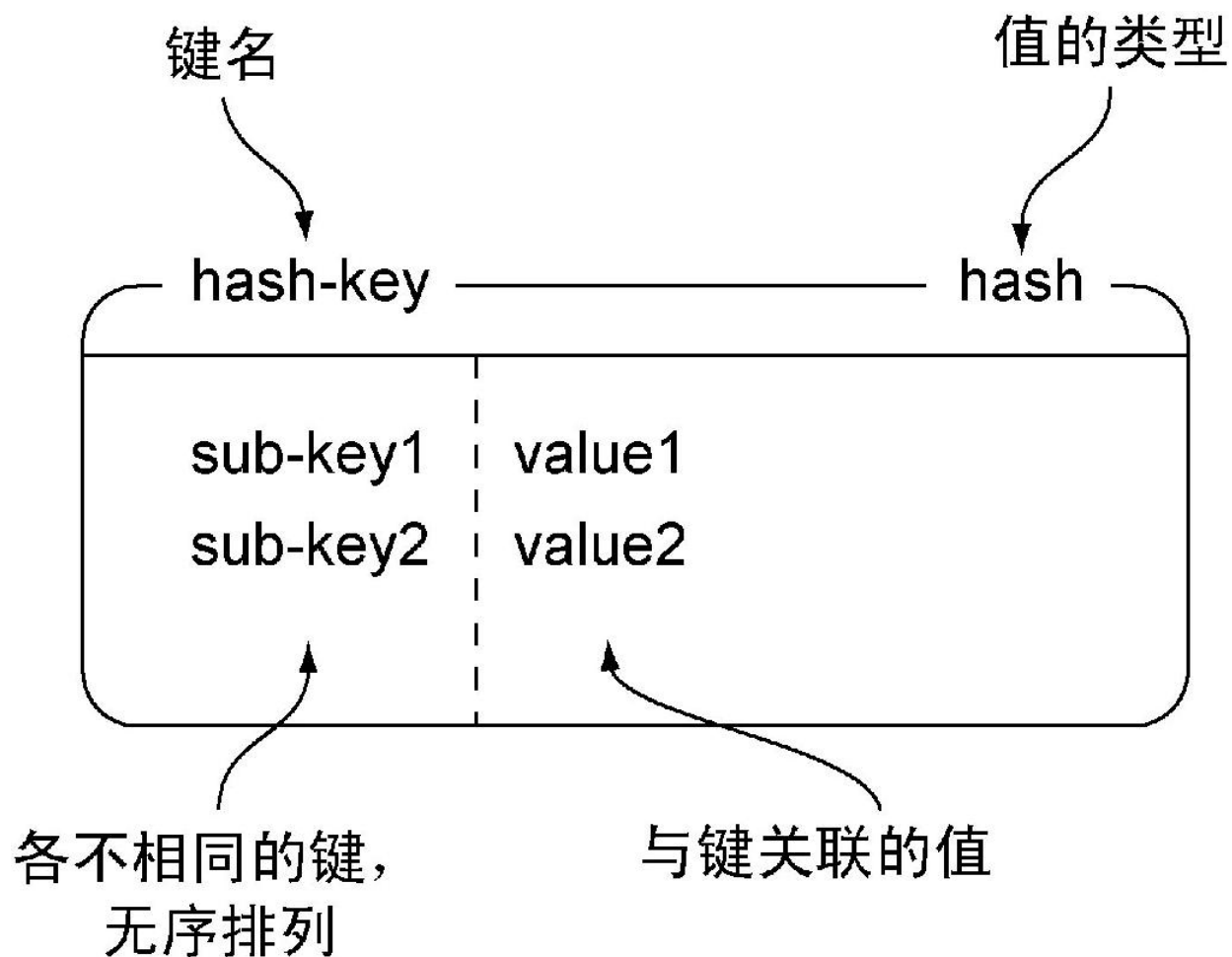


图1-4 hash-key的数据结构

Redis 1-4 数据结构

Redis 1-6 数据结构

1-4 HSET HGET HGETALL HDEL

```

redis 127.0.0.1:6379> hset hash-key sub-key1 value1
(integer) 1
redis 127.0.0.1:6379> hset hash-key sub-key2 value2
(integer) 1
redis 127.0.0.1:6379> hset hash-key sub-key1 value1
(integer) 0
redis 127.0.0.1:6379> hgetall hash-key
1) "sub-key1"
2) "value1"
3) "sub-key2"
4) "value2"
redis 127.0.0.1:6379> hdel hash-key sub-key2
(integer) 1
redis 127.0.0.1:6379> hdel hash-key sub-key2
(integer) 0
redis 127.0.0.1:6379> hget hash-key sub-key1
"value1"
redis 127.0.0.1:6379> hgetall hash-key
1) "sub-key1"
2) "value1"

```

在尝试添加键值对到散列的时候，命令会返回一个值来表示给定的键是否已经存在于散列里面。

在获取散列包含的所有键值对时，Python 客户端会把整个散列转换成一个 Python 字典。

在删除键值对的时候，命令会返回一个值来表示给定的键在移除之前是否存在于散列里面。

从散列里面获取某个键的值。

图 1-6 散列

命令	返回值
HSET	成功返回 1，失败返回 0
HGET	返回键对应的值
HGETALL	返回所有键值对
HDEL	成功返回删除的键数，失败返回 0

Redis 是一个开源的、高性能的、分布式的键值数据库。它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希等。Redis 还支持持久化、主从复制、集群等功能。Redis 的官方网站是 <http://redis.io>。

fieldRedis5

1.2.5 Redis

member
scoreRedis
1-5

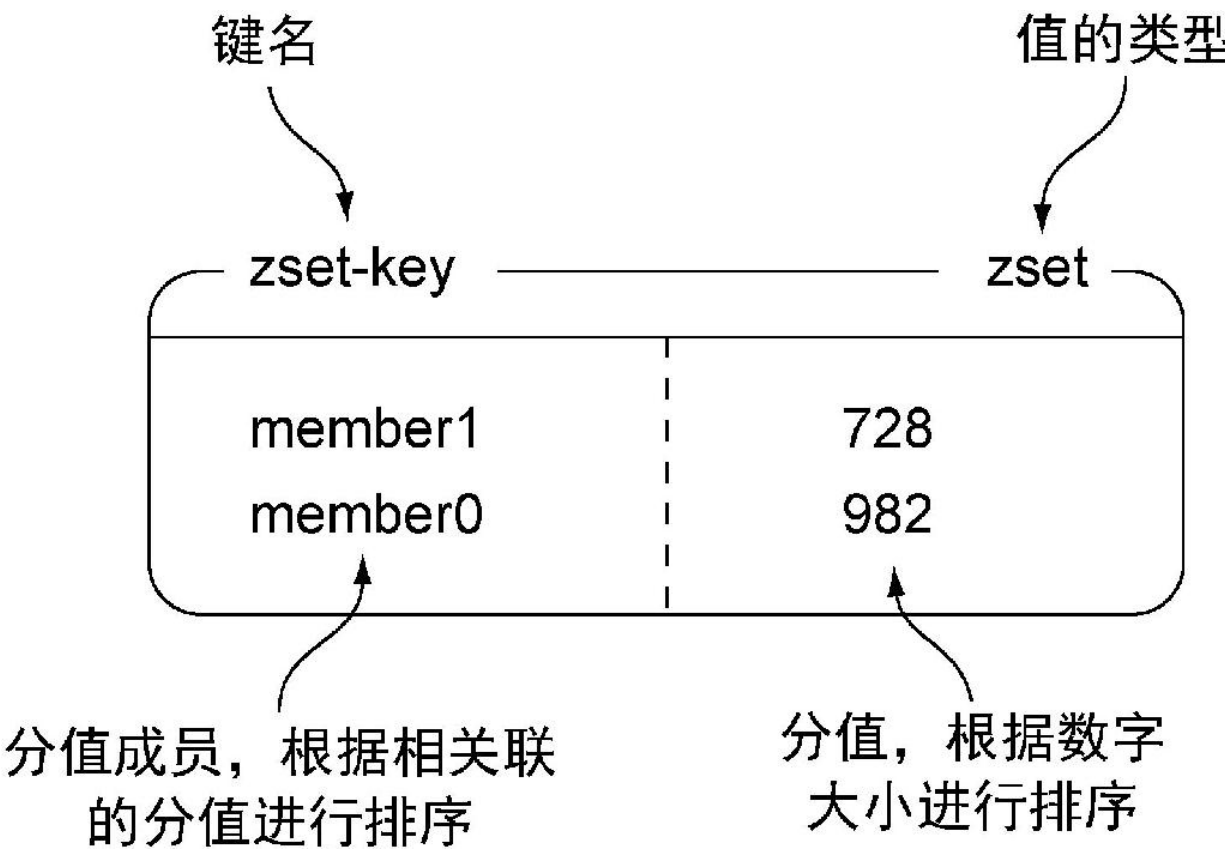


图1-5 zset-key

Redis有序集合1-5

Redis有序集合1-7

1-5 ZADD ZRANGE ZRANGEBYSCORE ZREM

```
redis 127.0.0.1:6379> zadd zset-key 728 member1
(integer) 1
redis 127.0.0.1:6379> zadd zset-key 982 member0
(integer) 1
redis 127.0.0.1:6379> zadd zset-key 982 member0
(integer) 0
redis 127.0.0.1:6379> zrange zset-key 0 -1 withscores
1) "member1"
2) "728"
3) "member0"
4) "982"
redis 127.0.0.1:6379> zrangebyscore zset-key 0 800 withscores
1) "member1"
2) "728"
redis 127.0.0.1:6379> zrem zset-key member1
(integer) 1
redis 127.0.0.1:6379> zrem zset-key member1
(integer) 0
redis 127.0.0.1:6379> zrange zset-key 0 -1 withscores
1) "member0"
2) "982"
```

在尝试向有序集合添加元素的时候，命令会返回新添加元素的数量。

在获取有序集合包含的所有元素时，多个元素会按照分值大小进行排序，并且 Python 客户端会将元素的分值转换成浮点数。

用户也可以根据分值来获取有序集合中的一部分元素。

在移除有序集合元素的时候，命令会返回被移除元素的数量。

1-7

命令	说明
ZADD	向有序集合添加元素
ZRANGE	返回有序集合中指定范围内的元素
ZRANGEBYSCORE	返回有序集合中指定分值范围内的元素

命令	命令
ZREM	删除有序集合中的成员

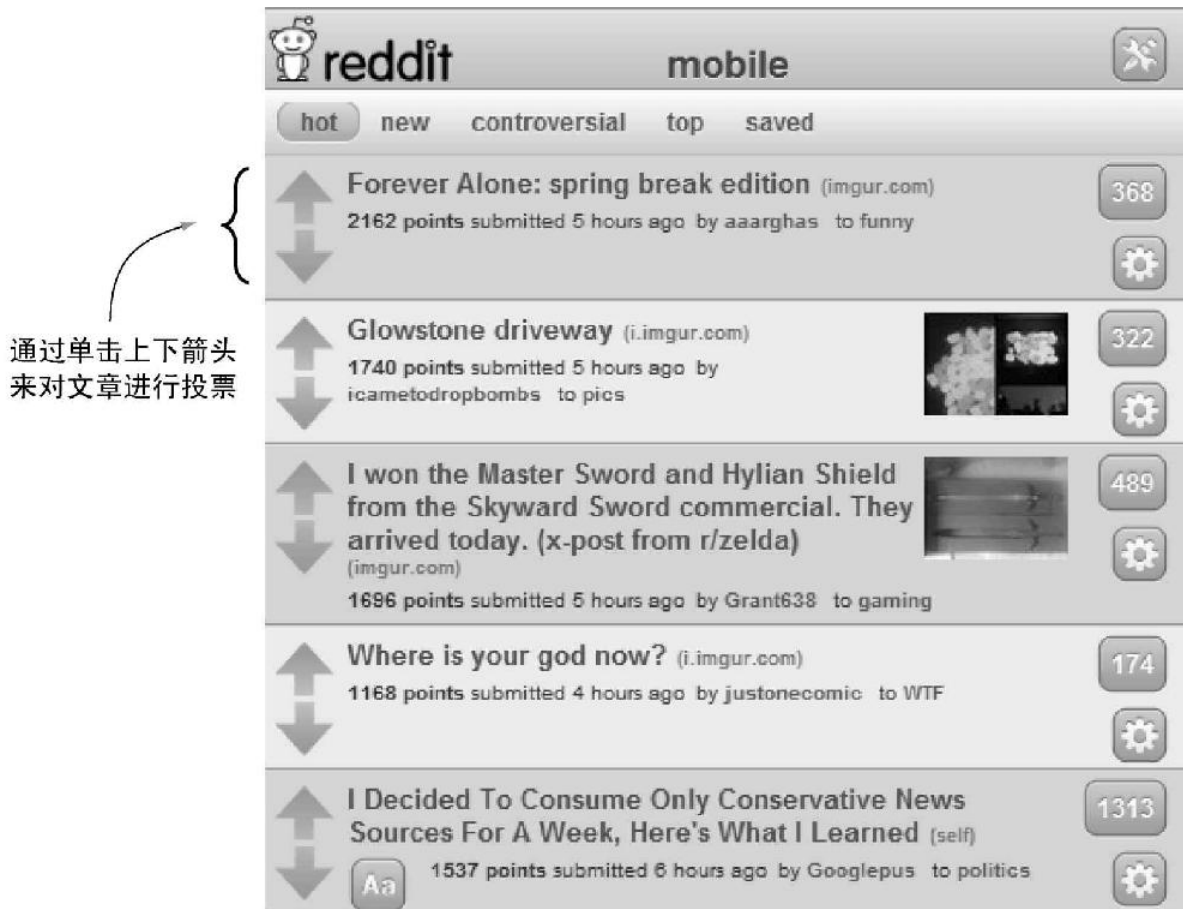
Redis有序集合的删除命令ZREM，用于删除有序集合中的成员。该命令的语法如下：

`ZREM key member` 删除有序集合 key 中的成员 member。

命令示例：

1.3 Redis

Redis 5
1-6
reddit 1-7
StackOverflow
Redis
Redis



1-6 Reddit

用户可以在浏览问题的同时，对问题以及问题已有的答案进行投票

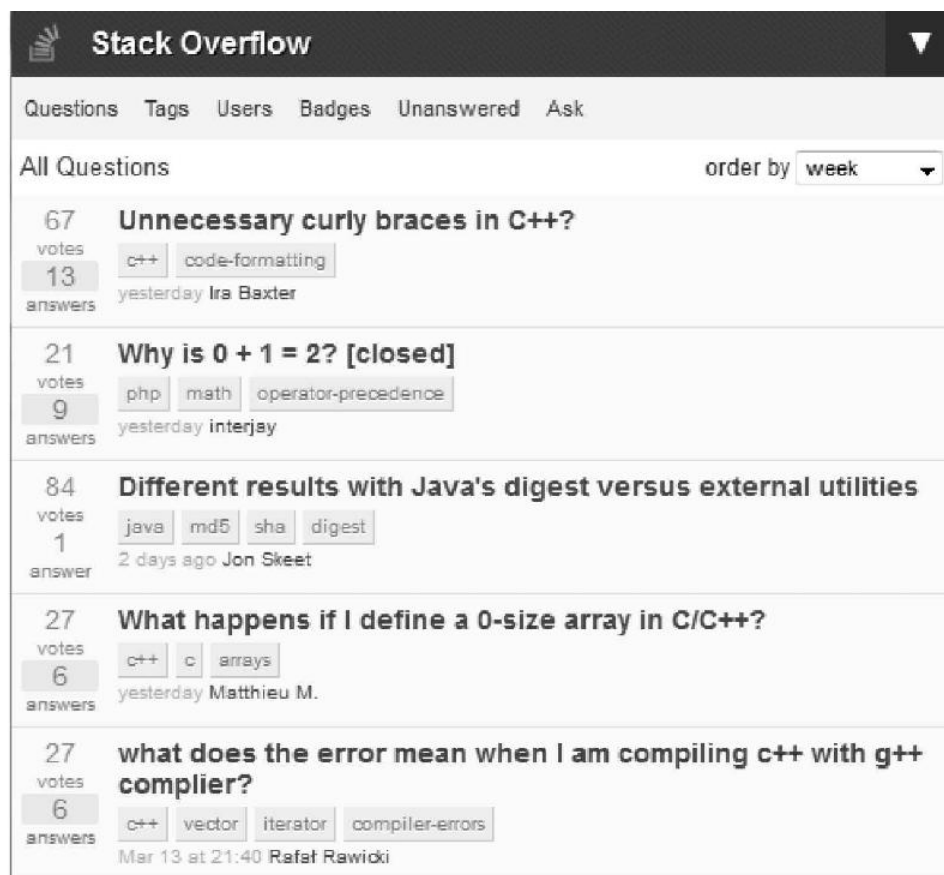


图1-7 StackOverflow网站截图

1.3.1 注册账号

注册账号非常简单，只需要填写用户名、密码、电子邮件地址、手机号码、生日、性别、职业、公司、行业、职位、教育背景、工作经历、项目经验、技能、兴趣爱好、个人简介等信息。注册成功后，用户将获得一个唯一的用户名和密码，可以用于登录和访问网站。此外，用户还可以选择是否接收邮件通知、是否公开个人资料、是否允许其他用户查看自己的活动记录等选项。

注册账号后，用户可以浏览问题、回答问题、投票、评论、私信、关注其他用户、创建个人资料页面、上传头像、设置个人资料、管理自己的问题、查看自己的活动记录、参与社区活动等。用户可以通过回答问题、获得高票数、获得徽章等方式来提升自己的声誉和影响力。

article:92617 ————— hash	
title	Go to statement considered harmful
link	http://goo.gl/kZUSu
poster	user:83271
time	1331382699.33
votes	528

图1-8 数据库中的文章数据

数据库中的文章数据存储在 Redis 的 Hash 类型中。图 1-8 展示了文章 ID 为 92617 的数据。在 Redis 中，文章数据存储在 namespace:article 命名空间中。namespace 是 Redis 数据库的命名空间，用于区分不同的数据库。在 Redis 中，命名空间是通过在键名前添加 namespace 来区分的。例如，在图 1-8 中，文章 ID 为 92617 的数据存储在 namespace:article:92617 键中。键中的 value 部分是一个 Hash 类型，其中包含文章标题、链接、发帖人、时间和投票数等信息。

在 Redis 中，文章 ID 是唯一的。每个文章都有一个唯一的 ID，用于标识该文章。在图 1-8 中，文章 ID 为 92617。在 Redis 中，文章 ID 是存储在 Hash 类型的键名中的。例如，在图 1-8 中，文章 ID 为 92617 的数据存储在 namespace:article:92617 键中。

1-9

time:	zset
article:100408	1332065417.47
article:100635	1332075503.49
article:100716	1332082035.26

根据发布时间排序文章的有序集合

score:	zset
article:100635	1332164063.49
article:100408	1332174713.47
article:100716	1332225027.26

根据评分排序文章的有序集合

1-9

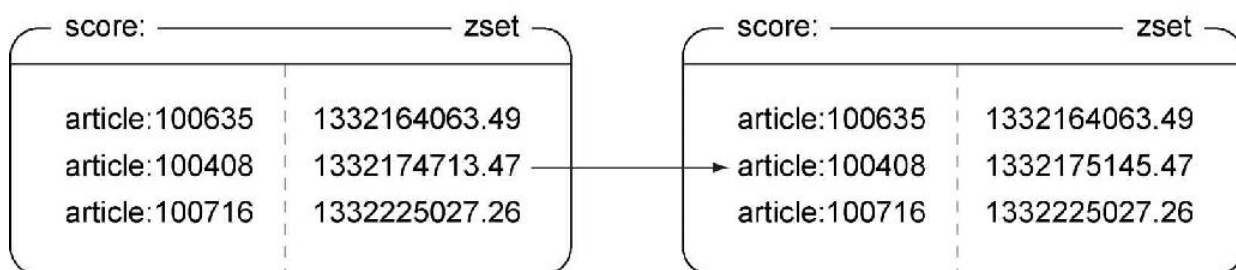
ID1-10

voted:100408	set
user:234487	
user:253378	
user:364680	
user:132097	
user:350917	
...	

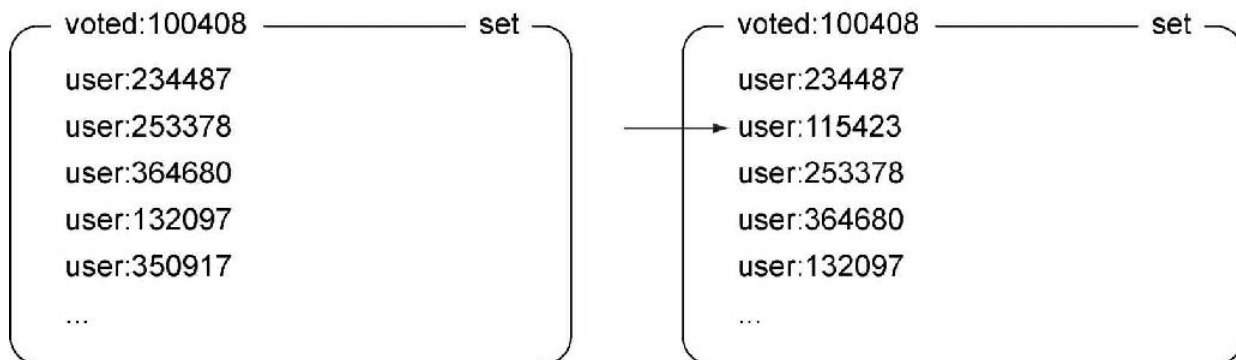
图1-10 对100408号文章的支持票

Redis中，我们使用ZSET来存储支持票。ZSET是一个有序集合，它的特点是集合中的成员都是唯一的，并且每个成员都有一个对应的分数（score）。在Redis中，ZSET的分数可以是浮点数，也可以是整数。在图1-10中，我们可以看到一个ZSET，它的成员是文章ID和它们的分数。

在图1-10中，我们可以看到一个ZSET，它的成员是文章ID和它们的分数。在图1-11中，我们可以看到一个SET，它的成员是用户ID。在图1-11中，我们可以看到一个SET，它的成员是用户ID。



100408号文章得到了一张新的支持票，它的评分增加了



115423号用户会被追加到对100408号文章的已投票用户名单里面

图1-11 对115423号用户的支持票

在Redis中，我们使用ZSET来存储支持票。ZSET是一个有序集合，它的特点是集合中的成员都是唯一的，并且每个成员都有一个对应的分数（score）。在Redis中，ZSET的分数可以是浮点数，也可以是整数。在图1-11中，我们可以看到一个ZSET，它的成员是文章ID和它们的分数。在图1-12中，我们可以看到一个SET，它的成员是用户ID。在图1-12中，我们可以看到一个SET，它的成员是用户ID。

SADD 1-6
 ZINCRBY 432
 HINCRBY
 HINCRBY 1-6

1-6 article_vote()

```
ONE_WEEK_IN_SECONDS = 7 * 86400
VOTE_SCORE = 432
```

准备好需要用到的常量。

```
def article_vote(conn, user, article):
    cutoff = time.time() - ONE_WEEK_IN_SECONDS
    if conn.zscore('time:', article) < cutoff:
        return

    article_id = article.partition(':')[ -1]
    if conn.sadd('voted:' + article_id, user):
        conn.zincrby('score:', article, VOTE_SCORE)
        conn.hincrby(article, 'votes', 1)
```

如果用户是第一次为这篇文章投票，那么增加这篇文章的投票数量和评分。

计算文章的投票截止时间。

检查是否还可以对文章进行投票（虽然使用散列也可以获取文章的发布时间，但有序集合返回的文章发布时间为浮点数，可以不进行转换直接使用）。

从 article:id 标识符（identifier）里面取出文章的 ID。

Redis 1-6 SADD
 ZINCRBY HINCRBY 3 4 Redis
 1-6

1.3.2

ID counter
 INCR SADD ID

EXPIRE Redis
HMSET ZADD
initial score 1-7
1-7

1-7 post_article()

```
def post_article(conn, user, title, link):
    article_id = str(conn.incr('article:'))

    voted = 'voted:' + article_id
    conn.sadd(voted, user)
    conn.expire(voted, ONE_WEEK_IN_SECONDS)

    now = time.time()
    article = 'article:' + article_id
    conn.hmset(article, {
        'title': title,
        'link': link,
        'poster': user,
        'time': now,
        'votes': 1,
    })

    conn.zadd('score:', article, now + VOTE_SCORE)
    conn.zadd('time:', article, now)

    return article_id
```

← 生成一个新的文章 ID。

将发布文章的用户添加到文章的已投票用户名单里面，然后将这个名单的过期时间设置为一周（第3章将对过期时间作更详细的介绍）。

将文章信息存储到一个散列里面。

将文章添加到根据发布时间排序的有序集合和根据评分排序的有序集合里面。

ZREVRANGE ID
ID HGETALL
ZREVRANGE “” ID 1-8

1-8 get_articles()

ARTICLES_PER_PAGE = 25

```
def get_articles(conn, page, order='score:'):
    start = (page-1) * ARTICLES_PER_PAGE
    end = start + ARTICLES_PER_PAGE - 1

    ids = conn.zrevrange(order, start, end)
    articles = []
    for id in ids:
        article_data = conn.hgetall(id)
        article_data['id'] = id
        articles.append(article_data)

    return articles
```

设置获取文章的起始索引和结束索引。

◀—— 获取多个文章 ID。

根据文章 ID 获取文章的
详细信息。

根据文章 ID 获取文章的详细信息。

Python 1-8 get_articles()

order score: Python “ ”
”
Python
<http://mng.bz/KM5x>

00
 000000group000000000000000000000000000000“0000”00
 0000“00”000000“Java00”00000000“Redis00”0000000000
 00000000000000000000000000

1.3.3

XX
XXID

1-9

1-9 add_remove_groups()

```
def add_remove_groups(conn, article_id, to_add=[], to_remove=[]):  
    article = 'article:' + article_id  
    for group in to_add:  
        conn.sadd('group:' + group, article)  
    for group in to_remove:  
        conn.srem('group:' + group, article)
```

构建存储文章信息的键名。

将文章添加到它所属的群组里面。

从群组里面移除文章。

Redis

paging

Redis ZINTERSTORE

combine

1

ZINTERSTORE

1-12

ZINTERSTORE

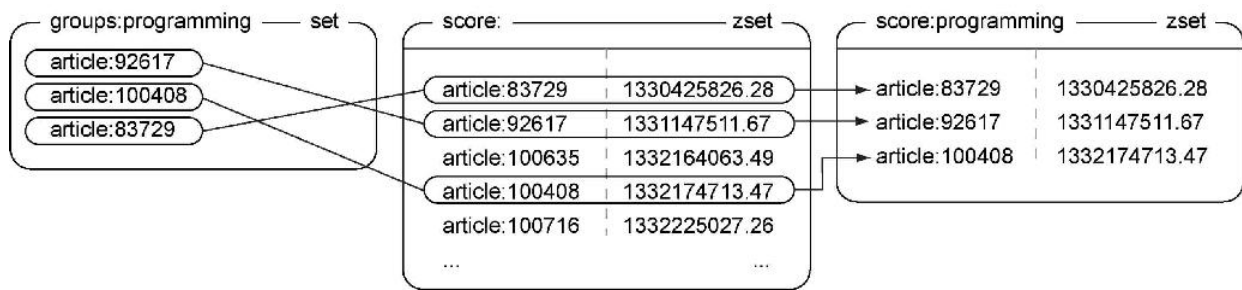


图1-12 groups:programming 和 score: 有序集合

```
score:programming 有序集合 groups:programming 有序集合 score: 有序集合
groups:programming 有序集合 1 有序集合 score: 有序集合 1 有序集合
有序集合 有序集合 score:programming 有序集合 score: 有序集合
```

有序集合 ZINTERSTORE 有序集合

有序集合 有序集合 有序集合 有序集合 有序集合 有序集合

ZINTERSTORE 有序集合 有序集合 有序集合 有序集合 有序集合 有序集合

有序 ZINTERSTORE 有序集合 Redis 有序集合 有序集合

有序 60 有序集合 get_articles() 有序集合 有序集合

1-10 有序集合 有序集合 有序集合

图1-10 get_group_articles()

```
def get_group_articles(conn, group, page, order='score:'):
    key = order + group
    if not conn.exists(key):
        conn.zinterstore(key,
            ['group:' + group, order],
            aggregate='max',
        )
        conn.expire(key, 60)
    return get_articles(conn, page, key)
```

为每个群组的
每种排列顺序
都创建一个键。

检查是否有已缓存的排序结果，
如果没有的话就现在进行排序。

让 Redis 在 60 秒之后自动删
除这个有序集合。

调用之前定义的 get_articles()
函数来进行分页并获取文章数据。

根据评分或
者发布时间，
对群组文章
进行排序。

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

1.4 资源

关于Redis的书籍和文档资源非常丰富，以下是一些推荐的资源：

关于Redis的书籍，推荐Manning出版的《Redis 设计与实现》（Redis in Depth），可以在<http://www.manning-sandbox.com/forum.jspa?forumID=809>找到购买链接。

关于Redis的文档，推荐Redis官方网站的文档，可以在<https://groups.google.com/d/forum/redis-db/>找到相关链接。

关于Redis的书籍，推荐Redis官方网站的文档，可以在<https://groups.google.com/d/forum/redis-db/>找到相关链接。

关于Redis的书籍，推荐Redis官方网站的文档，可以在<https://groups.google.com/d/forum/redis-db/>找到相关链接。

关于Redis的书籍，推荐Redis官方网站的文档，可以在<https://groups.google.com/d/forum/redis-db/>找到相关链接。

1.5 Redis

Redis 是一个开源的、基于内存的、支持持久化的数据库。它支持多种数据类型，如字符串、列表、集合、有序集合、哈希等。Redis 还支持主从复制、哨兵、集群等功能。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。

Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。

Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。

Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。

① Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。Redis 的部署和配置相对简单，且性能优秀，适用于各种场景。

② memcachedRedis3
Redis
RedisRedis
RedisRedisRedisRedisRedis
memcached

ePUBw.COM ePUBw.COM

2 Redis Web

- cookie
- cookie
-
-
-

1 Redis
Web
Redis 3 Redis

Web HTTP
service Web

1 request

2 handler

3

4. 在模板渲染前，调用 `template.render()`

5. 在响应生成前，调用 `response.render()`

在 5. 中，我们调用了 `Web.render()`，这里我们调用了 `Web.render()`，
在 `stateless` 中，我们调用了 `Web.render()`，
在 `fail` 中，我们调用了 `Web.render()`，
在 `Redis` 中，我们调用了 `Redis.render()`，
在 `Redis` 中，我们调用了 `Redis.render()`

在 `Fake Web Retailer` 中，我们调用了 `Fake Web Retailer.render()`，
在 `500` 中，我们调用了 `500.render()`，
在 `1` 中，我们调用了 `1.render()`，
在 `10` 中，我们调用了 `10.render()`，
在 `Fake Web Retailer` 中，我们调用了 `Fake Web Retailer.render()`，
在 `GB` 中，我们调用了 `GB.render()`，
在 `Redis` 中，我们调用了 `Redis.render()`，
在 `Redis` 中，我们调用了 `Redis.render()`

在 `Redis` 中，我们调用了 `Redis.render()`，
在 `Redis` 中，我们调用了 `Redis.render()`，
在 `Redis` 中，我们调用了 `Redis.render()`

在 `Redis` 中，我们调用了 `Redis.render()`，
在 `session` 中，我们调用了 `session.render()`

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

2.1 cookie

cookie
cookie
cookie
signed cookie token cookie

cookie ID
cookie
cookie

cookie cookie
2-1 cookie cookie
cookie

2-1 cookie cookie

cookie		
cookie	cookie cookie cookie additional infomation cookie	

cookie ID	Value	Domain
Cookie ID	Cookie Value Cookie Value	Cookie Domain Cookie Domain Cookie Domain

Fake Web Retailer Cookie ID
 cookie ID entry Fake
 Web Retailer
 ID

ID
 ID
 ID
 200 2000 ID
 ID

Fake Web Retailer ID — ID 1200
 6000 ID 10 ID
 Redis cookie ID cookie ID

cookie ID 2-1 cookie

2-1 check_token()

```
def check_token(conn, token):  
    return conn.hget('login:', token)
```

尝试获取并返回令牌对应的用户。

25 2-2

2-2 update_token()

```
def update_token(conn, token, user, item=None):  
    timestamp = time.time()  # 获取当前时间戳。  
    conn.hset('login:', token, user)  
    conn.zadd('recent:', token, timestamp)  
    if item:  
        conn.zadd('viewed:' + token, item, timestamp)  
        conn.zremrangebyrank('viewed:' + token, 0, -26)  
        # 记录令牌最后一次出现的时间。  
        # 移除旧的记录，只保留用户最近浏览过的 25 个商品。
```

记录用户浏览过的商品。

update_token() update_token() 20 000 Fake Web Retailer 6000 3

000 000/86 400<58
24×3600=86 400
000 000/86 400<58
60
10 000
60 000
150
1000

2-3
cron job
6.3
while not QUIT:

Python
(*vtokens)
unpack
Python
<http://mng.bz/8I7W>

Redis
Redis
cookie
Redis EXPIRE
Redis

1000

[illegible]

```

Redis
RedisWeb
Rediscookie

```

□□□□
ePUBw.COM
□□□□
ePUBw.COM
□□□□

[illegible]

2.2 Redis

Netscape 20 90 cookie cookie
cookie cookie web
retailer cookie

cookie — cookie
validate cookie cookie
cookie cookie
cookie

Redis cookie
Redis cookie cookie ID

ID
Web
0 ID

02-4 add_to_cart()

2-4 add_to_cart()

```
def add_to_cart(conn, session, item, count):
    if count <= 0:
        conn.hrem('cart:' + session, item)
    else:
        conn.hset('cart:' + session, item, count)
```

从购物车里面移除指定的商品。

将指定的商品添加到购物车。

2-5

2-5 clean_full_sessions()

```
def clean_full_sessions(conn):
    while not QUIT:
        size = conn.zcard('recent:')
        if size <= LIMIT:
            time.sleep(1)
            continue
        end_index = min(size - LIMIT, 100)
        sessions = conn.zrange('recent:', 0, end_index-1)
        session_keys = []
        for sess in sessions:
            session_keys.append('viewed:' + sess)
            session_keys.append('cart:' + sess)
        conn.delete(*session_keys)
        conn.hdel('login:', *sessions)
        conn.zrem('recent:', *sessions)
```

新增加的这行代码用于删除旧会话对应用户的购物车。

Redis

“X%”“
”

cookiecookieRedis
Web

ePUBw.COM ePUBw.COM

2.3 项目

项目使用模板引擎（templating language）来生成HTML页面。项目使用Web框架（framework）来处理HTTP请求和响应。项目使用JavaScript来生成动态内容。

项目Fake Web Retailer是一个模拟在线零售商的项目。项目使用Python语言开发。项目使用Django框架。项目使用Redis数据库。项目使用Celery任务队列。项目使用Sentry错误追踪。项目使用Gunicorn作为Web服务器。项目使用Nginx作为反向代理。项目使用Let's Encrypt证书。项目使用Docker容器化。项目使用Kubernetes编排。项目使用Prometheus监控。项目使用Grafana可视化。项目使用Jenkins持续集成。项目使用Ansible配置管理。项目使用Terraform基础设施即代码。项目使用Vault密钥管理。项目使用Consul服务发现。项目使用Elasticsearch搜索。项目使用Kibana可视化。项目使用Logstash日志处理。项目使用Fluentd日志处理。项目使用Kafka消息队列。项目使用RabbitMQ消息队列。项目使用Apache Spark大数据处理。项目使用Hadoop大数据存储。项目使用Pig大数据处理。项目使用MapReduce大数据处理。项目使用Hive大数据处理。项目使用Presto大数据处理。项目使用Trino大数据处理。项目使用Doris大数据处理。项目使用ClickHouse大数据处理。项目使用Flink大数据处理。项目使用Kudu大数据处理。项目使用HBase大数据处理。项目使用Cassandra大数据处理。项目使用ScyllaDB大数据处理。项目使用MongoDB大数据处理。项目使用CouchDB大数据处理。项目使用Neo4j大数据处理。项目使用GraphDB大数据处理。项目使用JanusGraph大数据处理。项目使用Apache Geophysics大数据处理。项目使用Apache Mahout大数据处理。项目使用Apache Pig大数据处理。项目使用Apache Tez大数据处理。项目使用Apache Yarn大数据处理。项目使用Apache Hama大数据处理。项目使用Apache Storm大数据处理。项目使用Apache Flink大数据处理。项目使用Apache Spark大数据处理。项目使用Apache DStream大数据处理。项目使用Apache Structured Streaming大数据处理。项目使用Apache Kudu大数据处理。项目使用Apache HBase大数据处理。项目使用Apache Cassandra大数据处理。项目使用Apache ScyllaDB大数据处理。项目使用Apache MongoDB大数据处理。项目使用Apache CouchDB大数据处理。项目使用Apache Neo4j大数据处理。项目使用Apache GraphDB大数据处理。项目使用Apache JanusGraph大数据处理。项目使用Apache Geophysics大数据处理。项目使用Apache Mahout大数据处理。项目使用Apache Pig大数据处理。项目使用Apache Tez大数据处理。项目使用Apache Yarn大数据处理。项目使用Apache Hama大数据处理。项目使用Apache Storm大数据处理。项目使用Apache Flink大数据处理。项目使用Apache Spark大数据处理。项目使用Apache DStream大数据处理。项目使用Apache Structured Streaming大数据处理。

项目Fake Web Retailer是一个模拟在线零售商的项目。项目使用Python语言开发。项目使用Django框架。项目使用Redis数据库。项目使用Celery任务队列。项目使用Sentry错误追踪。项目使用Gunicorn作为Web服务器。项目使用Nginx作为反向代理。项目使用Let's Encrypt证书。项目使用Docker容器化。项目使用Kubernetes编排。项目使用Prometheus监控。项目使用Grafana可视化。项目使用Jenkins持续集成。项目使用Ansible配置管理。项目使用Terraform基础设施即代码。项目使用Vault密钥管理。项目使用Consul服务发现。项目使用Elasticsearch搜索。项目使用Kibana可视化。项目使用Logstash日志处理。项目使用Fluentd日志处理。项目使用Kafka消息队列。项目使用RabbitMQ消息队列。项目使用Apache Spark大数据处理。项目使用Hadoop大数据存储。项目使用Pig大数据处理。项目使用MapReduce大数据处理。项目使用Hive大数据处理。项目使用Presto大数据处理。项目使用Trino大数据处理。项目使用Doris大数据处理。项目使用ClickHouse大数据处理。项目使用Flink大数据处理。项目使用Kudu大数据处理。项目使用HBase大数据处理。项目使用Cassandra大数据处理。项目使用ScyllaDB大数据处理。项目使用MongoDB大数据处理。项目使用CouchDB大数据处理。项目使用Neo4j大数据处理。项目使用GraphDB大数据处理。项目使用JanusGraph大数据处理。项目使用Apache Geophysics大数据处理。项目使用Apache Mahout大数据处理。项目使用Apache Pig大数据处理。项目使用Apache Tez大数据处理。项目使用Apache Yarn大数据处理。项目使用Apache Hama大数据处理。项目使用Apache Storm大数据处理。项目使用Apache Flink大数据处理。项目使用Apache Spark大数据处理。项目使用Apache DStream大数据处理。项目使用Apache Structured Streaming大数据处理。

项目使用Python语言开发。项目使用Django框架。项目使用Redis数据库。项目使用Celery任务队列。项目使用Sentry错误追踪。项目使用Gunicorn作为Web服务器。项目使用Nginx作为反向代理。项目使用Let's Encrypt证书。项目使用Docker容器化。项目使用Kubernetes编排。项目使用Prometheus监控。项目使用Grafana可视化。项目使用Jenkins持续集成。项目使用Ansible配置管理。项目使用Terraform基础设施即代码。项目使用Vault密钥管理。项目使用Consul服务发现。项目使用Elasticsearch搜索。项目使用Kibana可视化。项目使用Logstash日志处理。项目使用Fluentd日志处理。项目使用Kafka消息队列。项目使用RabbitMQ消息队列。项目使用Apache Spark大数据处理。项目使用Hadoop大数据存储。项目使用Pig大数据处理。项目使用MapReduce大数据处理。项目使用Hive大数据处理。项目使用Presto大数据处理。项目使用Trino大数据处理。项目使用Doris大数据处理。项目使用ClickHouse大数据处理。项目使用Flink大数据处理。项目使用Kudu大数据处理。项目使用HBase大数据处理。项目使用Cassandra大数据处理。项目使用ScyllaDB大数据处理。项目使用MongoDB大数据处理。项目使用CouchDB大数据处理。项目使用Neo4j大数据处理。项目使用GraphDB大数据处理。项目使用JanusGraph大数据处理。项目使用Apache Geophysics大数据处理。项目使用Apache Mahout大数据处理。项目使用Apache Pig大数据处理。项目使用Apache Tez大数据处理。项目使用Apache Yarn大数据处理。项目使用Apache Hama大数据处理。项目使用Apache Storm大数据处理。项目使用Apache Flink大数据处理。项目使用Apache Spark大数据处理。项目使用Apache DStream大数据处理。项目使用Apache Structured Streaming大数据处理。

项目2-6 `cache_request()`



❶ Fake Web Retailer ❷ 95% ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 2-6 ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 Redis ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 Redis ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿

❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 Redis ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 Redis ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿
 ❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽ ❾ ❿ ⓫ ⓬ ⓭ ⓮ ⓯ ⓰ ⓱ ⓲ ⓳ ⓴ ⓵ ⓶ ⓷ ⓸ ⓹ ⓺ ⓻ ⓼ ⓽ ⓾ ⓿

❶ ❷ ❸ ❹ ePUBw.COM ❶ ❷ ❸ ❹ ePUBw.COM ❶ ❷ ❸ ❹
 ❶ ❷ ❸ ❹ ❶ ❷ ❸ ❹ ❶ ❷ ❸ ❹ ❶ ❷ ❸ ❹

2.4 数据库

数据库是应用程序的重要组成部分。在本章中，我们将使用Redis数据库。Redis是一个开源的、基于内存的、支持多种数据类型的数据库。它支持字符串、列表、集合、有序集合、哈希表、位图等数据类型。Redis还支持持久化、主从复制、集群等功能。

Fake Web Retailer是一个基于Redis的Web应用。它模拟了一个在线零售商，用户可以通过AJAX接口与服务器进行交互。用户可以通过搜索、浏览商品、加入购物车、下单等操作。该应用使用Redis作为数据库，存储商品信息、用户购物车、订单信息等。

在本章中，我们将介绍如何搭建Fake Web Retailer应用。我们将使用Redis数据库，并使用AJAX接口与服务器进行交互。我们将使用jQuery库来简化JavaScript代码。我们将使用Bootstrap框架来美化界面。我们将使用Node.js和Express.js来搭建服务器。我们将使用Redis数据库来存储数据。

在本章中，我们将介绍如何搭建Fake Web Retailer应用。我们将使用Redis数据库，并使用AJAX接口与服务器进行交互。我们将使用jQuery库来简化JavaScript代码。我们将使用Bootstrap框架来美化界面。我们将使用Node.js和Express.js来搭建服务器。我们将使用Redis数据库来存储数据。

我们将使用Redis数据库来存储数据。Redis是一个开源的、基于内存的、支持多种数据类型的数据库。它支持字符串、列表、集合、有序集合、哈希表、位图等数据类型。Redis还支持持久化、主从复制、集群等功能。

我们将使用AJAX接口与服务器进行交互。AJAX是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。它使用XMLHttpRequest对象来与服务器进行通信。

我们将使用jQuery库来简化JavaScript代码。jQuery是一个轻量级的JavaScript库，它简化了HTML文档遍历、事件处理、动画以及Ajax请求。它使用简洁的语法来操作DOM元素。

我们将使用Bootstrap框架来美化界面。Bootstrap是一个流行的前端框架，它提供了预定义的可响应式布局、组件和工具类。它使用CSS和JavaScript来实现。

我们将使用Node.js和Express.js来搭建服务器。Node.js是一个基于Chrome V8引擎的JavaScript运行环境。Express.js是一个基于Node.js的Web应用框架，它提供了简洁的API来处理路由、中间件、模板引擎等。

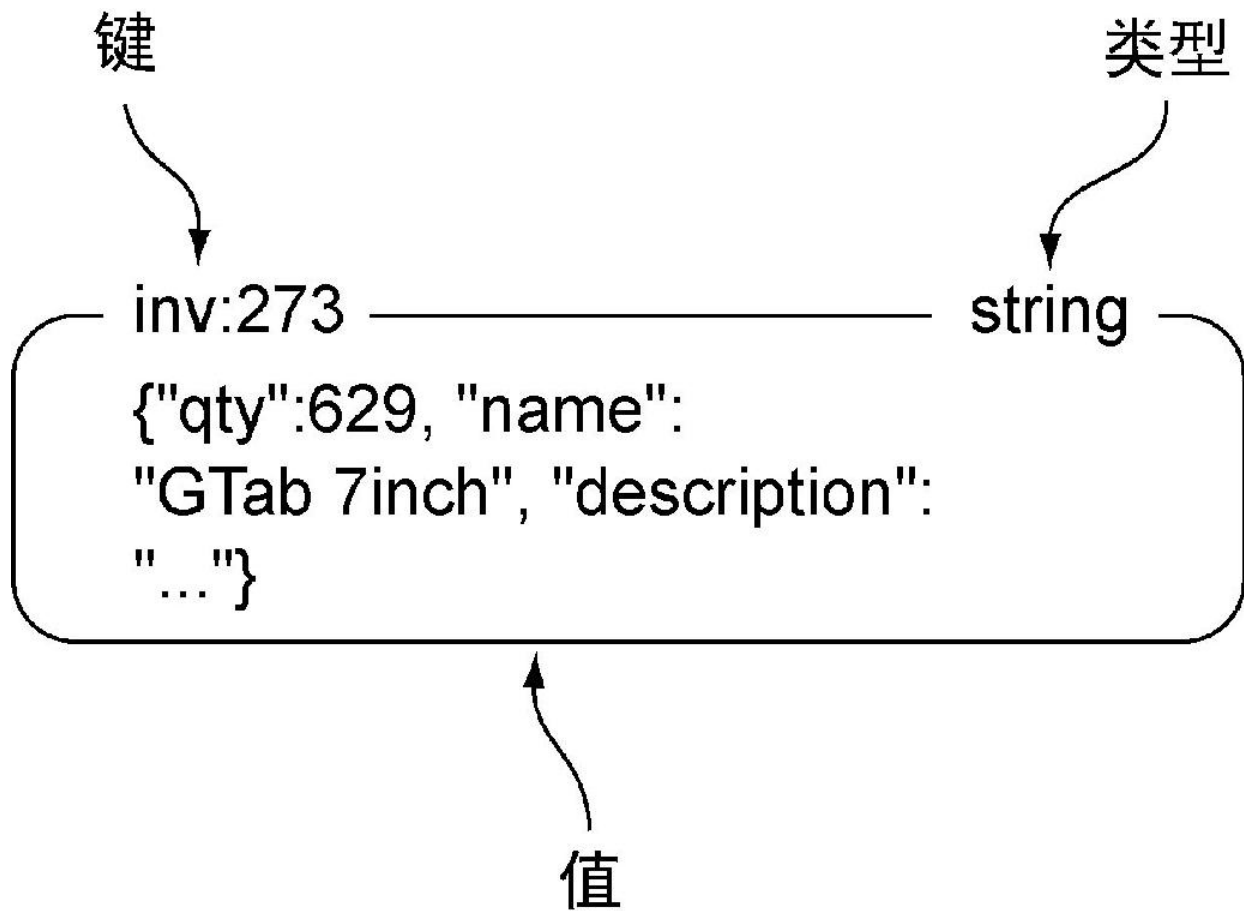


图2-1 Redis数据库的键值对

Redis数据库的键值对可以存储在内存中，也可以存储在磁盘上。Redis数据库的键值对可以存储在内存中，也可以存储在磁盘上。Redis数据库的键值对可以存储在内存中，也可以存储在磁盘上。

Redis数据库的键值对可以存储在内存中，也可以存储在磁盘上。Redis数据库的键值对可以存储在内存中，也可以存储在磁盘上。Redis数据库的键值对可以存储在内存中，也可以存储在磁盘上。

Redis 数据库的过期时间。Redis 数据库的过期时间默认为 0，即永不过期。Redis 数据库的过期时间可以通过 Redis 命令 SETEX 来设置。SETEX 命令的语法如下：

```
SETEX key seconds value
```

其中，key 是 Redis 数据库的键名，seconds 是过期时间（以秒为单位），value 是键值。例如，要将键名为 mykey 的 Redis 数据库键值设置为 123，并设置过期时间为 50 秒，可以使用以下命令：

```
SETEX mykey 50 123
```

Redis 数据库的过期时间也可以通过 Redis 命令 EXPIRE 来设置。EXPIRE 命令的语法如下：

```
EXPIRE key seconds
```

其中，key 是 Redis 数据库的键名，seconds 是过期时间（以秒为单位）。例如，要将键名为 mykey 的 Redis 数据库键值的过期时间设置为 50 秒，可以使用以下命令：

```
EXPIRE mykey 50
```

Redis 数据库的过期时间也可以通过 Redis 命令 TTL 来查看。TTL 命令的语法如下：

```
TTL key
```

其中，key 是 Redis 数据库的键名。例如，要查看键名为 mykey 的 Redis 数据库键值的过期时间，可以使用以下命令：

```
TTL mykey
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIRE 来设置。PEXPIRE 命令的语法如下：

```
PEXPIRE key milliseconds
```

其中，key 是 Redis 数据库的键名，milliseconds 是过期时间（以毫秒为单位）。例如，要将键名为 mykey 的 Redis 数据库键值的过期时间设置为 50 毫秒，可以使用以下命令：

```
PEXPIRE mykey 50
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIRETIME 来查看。PEXPIRETIME 命令的语法如下：

```
PEXPIRETIME key
```

其中，key 是 Redis 数据库的键名。例如，要查看键名为 mykey 的 Redis 数据库键值的过期时间（以毫秒为单位），可以使用以下命令：

```
PEXPIRETIME mykey
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIREAT 来设置。PEXPIREAT 命令的语法如下：

```
PEXPIREAT key unix_timestamp
```

其中，key 是 Redis 数据库的键名，unix_timestamp 是过期时间（以 Unix 时间戳为单位）。例如，要将键名为 mykey 的 Redis 数据库键值的过期时间设置为 2025 年 1 月 1 日 00:00:00，可以使用以下命令：

```
PEXPIREAT mykey 1735689600
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIREATIME 来查看。PEXPIREATIME 命令的语法如下：

```
PEXPIREATIME key
```

其中，key 是 Redis 数据库的键名。例如，要查看键名为 mykey 的 Redis 数据库键值的过期时间（以 Unix 时间戳为单位），可以使用以下命令：

```
PEXPIREATIME mykey
```

图 2-8 缓存数据行 cache_rows()

```
def cache_rows(conn):  
    while not QUIT:  
        next = conn.zrange('schedule:', 0, 0, withscores=True)  
        now = time.time()  
        if not next or next[0][1] > now:  
            time.sleep(.05)  
            continue  
        row_id = next[0][0]  
  
        delay = conn.zscore('delay:', row_id)  
        if delay <= 0:  
            conn.zrem('delay:', row_id)  
            conn.zrem('schedule:', row_id)  
            conn.delete('inv:' + row_id)  
            continue  
  
        row = Inventory.get(row_id)  
        conn.zadd('schedule:', row_id, now + delay)  
        conn.set('inv:' + row_id, json.dumps(row.to_dict()))
```

更新调度时间并设置缓存值。

尝试获取下一个需要被缓存的数据行以及该行的调度时间戳，命令会返回一个包含零个或一个元组 (tuple) 的列表。

暂时没有行需要被缓存，休眠 50 毫秒后重试。

提前获取下一次调度的延迟时间。

不必再缓存这个行，将它从缓存中移除。

读取数据行。

Redis 数据库的过期时间默认为 0，即永不过期。Redis 数据库的过期时间可以通过 Redis 命令 SETEX 来设置。SETEX 命令的语法如下：

```
SETEX key seconds value
```

其中，key 是 Redis 数据库的键名，seconds 是过期时间（以秒为单位），value 是键值。例如，要将键名为 mykey 的 Redis 数据库键值设置为 123，并设置过期时间为 50 秒，可以使用以下命令：

```
SETEX mykey 50 123
```

Redis 数据库的过期时间也可以通过 Redis 命令 EXPIRE 来设置。EXPIRE 命令的语法如下：

```
EXPIRE key seconds
```

其中，key 是 Redis 数据库的键名，seconds 是过期时间（以秒为单位）。例如，要将键名为 mykey 的 Redis 数据库键值的过期时间设置为 50 秒，可以使用以下命令：

```
EXPIRE mykey 50
```

Redis 数据库的过期时间也可以通过 Redis 命令 TTL 来查看。TTL 命令的语法如下：

```
TTL key
```

其中，key 是 Redis 数据库的键名。例如，要查看键名为 mykey 的 Redis 数据库键值的过期时间，可以使用以下命令：

```
TTL mykey
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIRE 来设置。PEXPIRE 命令的语法如下：

```
PEXPIRE key milliseconds
```

其中，key 是 Redis 数据库的键名，milliseconds 是过期时间（以毫秒为单位）。例如，要将键名为 mykey 的 Redis 数据库键值的过期时间设置为 50 毫秒，可以使用以下命令：

```
PEXPIRE mykey 50
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIRETIME 来查看。PEXPIRETIME 命令的语法如下：

```
PEXPIRETIME key
```

其中，key 是 Redis 数据库的键名。例如，要查看键名为 mykey 的 Redis 数据库键值的过期时间（以毫秒为单位），可以使用以下命令：

```
PEXPIRETIME mykey
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIREAT 来设置。PEXPIREAT 命令的语法如下：

```
PEXPIREAT key unix_timestamp
```

其中，key 是 Redis 数据库的键名，unix_timestamp 是过期时间（以 Unix 时间戳为单位）。例如，要将键名为 mykey 的 Redis 数据库键值的过期时间设置为 2025 年 1 月 1 日 00:00:00，可以使用以下命令：

```
PEXPIREAT mykey 1735689600
```

Redis 数据库的过期时间也可以通过 Redis 命令 PEXPIREATIME 来查看。PEXPIREATIME 命令的语法如下：

```
PEXPIREATIME key
```

其中，key 是 Redis 数据库的键名。例如，要查看键名为 mykey 的 Redis 数据库键值的过期时间（以 Unix 时间戳为单位），可以使用以下命令：

```
PEXPIREATIME mykey
```

Redis

[ePUBw.COM](#) [ePUBw.COM](#)

2.5 数据库

数据库是系统的重要组成部分，它负责存储和管理数据。在本章中，我们将介绍如何使用 Redis 数据库来存储用户登录信息。

在 2.1 和 2.2 节中，我们分别介绍了如何设置 Redis 数据库以及如何使用 Redis 数据库来存储用户登录信息。在 2.3 节中，我们将介绍如何使用 Redis 数据库来存储用户登录信息。Web 数据库和 Fake Web Retailer 数据库分别存储 100 000 条和 10 000 条数据。

在 2.1 节中，我们介绍了如何使用 Redis 数据库来存储用户登录信息。在 2.2 节中，我们介绍了如何使用 Redis 数据库来存储用户登录信息。在 2.3 节中，我们将介绍如何使用 Redis 数据库来存储用户登录信息。update_token() 函数用于更新用户登录信息。

2-9 数据库 update_token()

```
def update_token(conn, token, user, item=None):
    timestamp = time.time()
    conn.hset('login:', token, user)
    conn.zadd('recent:', token, timestamp)
    if item:
        conn.zadd('viewed:' + token, item, timestamp)
        conn.zremrangebyrank('viewed:' + token, 0, -26)
        conn.zincrby('viewed:', item, -1)
```

这行代码是新添加的。

在代码清单 2-10 中，我们使用 `zremrangebyrank` 方法删除所有排名在 20 000 名之后的商品。然后，我们使用 `zinterstore` 方法将浏览次数降低为原来的一半。最后，我们使用 `time.sleep` 方法在 5 分钟之后再次执行这个操作。

在代码清单 2-11 中，我们使用 `can_cache` 方法检查商品是否可以缓存。如果商品可以缓存，我们将其添加到缓存中。如果商品不能缓存，我们将其从缓存中删除。最后，我们使用 `zinterstore` 方法将浏览次数降低为原来的一半。

代码清单 2-10 `rescale_viewed()`

```
def rescale_viewed(conn):  
    while not QUIT:  
        conn.zremrangebyrank('viewed:', 0, -20001)  
        conn.zinterstore('viewed:', {'viewed:': .5})  
        time.sleep(300)
```

删除所有排名在 20 000 名之后的商品。

将浏览次数降低为原来的一半。

5 分钟之后再次执行这个操作。

在代码清单 2-11 中，我们使用 `can_cache` 方法检查商品是否可以缓存。如果商品可以缓存，我们将其添加到缓存中。如果商品不能缓存，我们将其从缓存中删除。最后，我们使用 `zinterstore` 方法将浏览次数降低为原来的一半。

代码清单 2-11 `can_cache()`

```

def can_cache(conn, request):
    item_id = extract_item_id(request)
    if not item_id or is_dynamic(request):
        return False
    rank = conn.zrank('viewed:', item_id)
    return rank is not None and rank < 10000

```

尝试从页面里面取出商品 ID。

检查这个页面能否被缓存以及这个页面是否为商品页面。

取得商品的浏览次数排名。

根据商品的浏览次数排名来判断是否需要缓存这个页面。

10 000
 Redis
 Edge Side Includes
 pre-optimize
 Redis

ePUBw.COM
ePUBw.COM

2.6 测试

测试 Fake Web Retailer 的 Web 页面功能

Web 页面功能测试

测试 cookie 功能

cookie 功能测试

测试 Redis 功能

Redis 功能测试

测试 Redis 功能

Redis 功能测试

测试 Redis 功能

① Fake Web Retailer 的 Web 页面功能

Web 页面功能测试

ePUBw.COM ePUBw.COM

测试

□□□□ □□□□

□□□□□□□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□□
□□□□□Redis□□□□□□□□□□□□□□□□□□□□Redis□□□□□□□□□□
□□

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□□□

3 Redis

-
-
-
-

12RedisRedis2

1

GETSETRedis

<http://redis.io/commands>.

Redis 2.4Redis 2.6 可以在A平台上运行Windows
Redis 2.4Redis 2.6Redis 2.4
Redis 2.6Lua11
PTTLPEXPIREPEXPIREAT
BITOPBITCOUNTRedis 2.6
RPUSHLPUSHSADDsREMhDELZADDZREMRedis 2.6
Redis 2.6

ePUBw.COMePUBw.COM

3.1 数据类型

Redis 1 和 2 版本支持 Redis 数据类型有字符串、哈希、列表、集合、有序集合、位图、超文本标记语言 (HTML) 和地理空间索引。Redis 3 版本支持

- 字符串 (string)
- 哈希 (hash)
- 列表 (list)

Redis 支持 increment 和 decrement 操作。Redis 支持 long integer (32 位和 64 位)、IEEE 754 double 和 Redis 支持的 Redis 数据类型。Redis 支持 Redis 数据类型。

Redis 支持 Redis 数据类型——Redis 支持 Redis 数据类型。Redis 支持 Redis 数据类型。Redis 支持 Redis 数据类型。

3-1 Redis 数据类型

3-1 Redis 命令

命令	命令格式
INCR	INCR key-name——自增1
DECR	DECR key-name——自减1
INCRBY	INCRBY key-name amount——自增amount
DECRBY	DECRBY key-name amount——自减amount
INCRBYFLOAT	INCRBYFLOAT key-name amount——自增amount amount为浮点数Redis 2.6开始支持

Redis 命令的格式为：command [key-name] [amount] [options]。其中，command 为命令名称，key-name 为键名，amount 为自增或自减的数值，options 为可选参数。Redis 命令的格式与 Redis 2.6 版本一致。

Redis 命令的格式与 Redis 2.6 版本一致。

Redis 命令的格式与 Redis 2.6 版本一致。

Redis 命令的格式与 Redis 2.6 版本一致。

Redis 命令的格式与 Redis 2.6 版本一致。

3-1 Redis INCR 和 DECR 命令

和自增操作一样，执行自减操作的函数也可以通过可选的参数来指定减量。

在尝试获取一个键的时候，命令将以字符串格式返回被存储的整数。

```
>>> conn = redis.Redis()
>>> conn.get('key')
>>> conn.incr('key')
1
>>> conn.incr('key', 15)
16
>>> conn.decr('key', 5)
11
>>> conn.get('key')
'11'
>>> conn.set('key', '13')
True
>>> conn.incr('key')
14
```

尝试获取一个不存在的键将得到一个 None 值，终端不会显示这个值。

我们既可以对不存在的键执行自增操作，也可以通过可选的参数来指定自增操作的增量。

即使在设置键时输入的值为字符串，但只要这个值可以被解释为整数，我们就可以把它当作整数来处理。

incr() Python Redis INCRBY incr() 1 Python Redis incrbyfloat() INCRBYFLOAT incrbyfloat() incr() Redis

Redis Redis 9 pack 3-2

3-2 Redis

命令	描述
APPEND	APPEND key-name value——将value追加到key-name的末尾

命令	命令格式
GETRANGE	GETRANGE key-name start end——返回key-name从start到end的字符串 返回的字符串长度是end-start+1
SETRANGE	SETRANGE key-name offset value——将key-name从start位置开始替换为value 返回OK
GETBIT	GETBIT key-name offset——返回key-name在offset位置的bit string 返回的字符串长度是offset+1
SETBIT	SETBIT key-name offset value——将key-name在offset位置的bit value 返回offset位置的value
BITCOUNT	BITCOUNT key-name [start end]——返回key-name从start到end的字符串中1的个数 返回的字符串长度是end-start+1
BITOP	BITOP operation dest-key key-name [key-name ...] ——对dest-key和key-name进行AND、OR、XOR、NOT等位运算 返回的字符串长度是dest-key的字符串长度

GETRANGE SUBSTR Redis GETRANGE

SUBSTR Python substr()

2.6 Redis getrange()

SETRANGE SETBIT

Redis null

GETRANGE GETBIT

0 3-2

3-2 Redis

APPEND 命令在
执行之后会返
回字符串当前
的长度。

对字符串执行范
围设置操作。

SETRANGE 命令在
执行之后同样会返
回字符串的当前总
长度。

SETRANGE 命令既可
以用于替换字符串里
已有的内容，又可以
用于增长字符串。

SETBIT 命令会返
回二进制位被设置
之前的值。

将字符串 'hello' 追加到目前并不存在的
'new-string-key' 键里。

Redis 的索引以 0 为开始，
在进行范围访问时，范围的
终点（endpoint）默认也包
含在这个范围之内。

字符串 'lo wo' 位于字符串
'hello world!' 的中间。

查看字符串当前的值。

前面执行的两个
SETRANGE 命令成功
地将字母 h 和 w 从原
来的小写改成了大写。

前面执行的 SETRANGE 命令移除
了字符串末尾的感叹号，并将更
多字符追加到了字符串末尾。

对超出字符串长度的二进
制位进行设置时，超出的
部分会被填充为空字节。

在对 Redis 存储的二进制位进
行解释（interpret）时，请记住
Redis 存储的二进制位是按照
偏移量从高到低排列的。

```
>>> conn.append('new-string-key', 'hello ')
6L
>>> conn.append('new-string-key', 'world!')
12L
>>> conn.substr('new-string-key', 3, 7)
'lo wo'
>>> conn.setrange('new-string-key', 0, 'H')
```

```
12
>>> conn.setrange('new-string-key', 6, 'W')
12
>>> conn.get('new-string-key')
'Hello World!'
```

```
>>> conn.setrange('new-string-key', 11, ' how are you?')
25
>>> conn.get('new-string-key')
'Hello World, how are you?'
>>> conn.setbit('another-key', 2, 1)
```

```
0
>>> conn.setbit('another-key', 7, 1)
0
>>> conn.get('another-key')
'!'
```

通过将第 2 个二进制位以及第 7 个二进制
位的值设置为 1，键的值将变为 '!'，也
就是编码为 33 的字符。

Redis 3.7.2 3 4 9

Redis

ePUBw.COM ePUBw.COM

3.2 列表

图3-1展示了Redis列表的底层实现。列表的底层实现是双向链表，每个节点包含一个指向下一个节点的指针，以及一个指向上一个节点的指针。列表的头部和尾部都指向第一个和最后一个节点。列表的头部和尾部都指向第一个和最后一个节点。

列表的底层实现是双向链表，每个节点包含一个指向下一个节点的指针，以及一个指向上一个节点的指针。列表的头部和尾部都指向第一个和最后一个节点。列表的头部和尾部都指向第一个和最后一个节点。

图3-3 列表命令

命令	命令格式
RPush	RPush key-name value [value ...]——向列表尾部添加元素
LPush	LPush key-name value [value ...]——向列表头部添加元素
RPop	RPop key-name——从列表尾部移除元素
LPop	LPop key-name——从列表头部移除元素
LIndex	LIndex key-name offset——返回列表中offset位置的元素
LRANGE	LRANGE key-name start end——返回列表中start和end之间的元素

命令	命令格式
LTRIM	<p>LTRIM key-name start end——将列表key-name中start和end之间的元素删除。</p> <p>命令格式：LTRIM key-name start end</p>

命令格式：LTRIM key-name start end

命令格式：LTRIM key-name start end

命令格式：LTRIM key-name start end

在向列表推入元素时，推入操作执行完毕之后会返回列表当前的长度。

```
>>> conn.rpush('list-key', 'last')
1L
>>> conn.lpush('list-key', 'first')
2L
>>> conn.rpush('list-key', 'new last')
3L
>>> conn.lrange('list-key', 0, -1)
['first', 'last', 'new last']
>>> conn.lpop('list-key')
'first'
>>> conn.lpop('list-key')
'last'
>>> conn.lrange('list-key', 0, -1)
['new last']
>>> conn.rpush('list-key', 'a', 'b', 'c')
4L
>>> conn.lrange('list-key', 0, -1)
['new last', 'a', 'b', 'c']
>>> conn.ltrim('list-key', 2, -1)
True
>>> conn.lrange('list-key', 0, -1)
['b', 'c']
```

可以很容易地对列表的两端执行推入操作。

从语义上来说，列表的左端为开头，右端为结尾。

通过重复地弹出列表左端的元素，可以按照从左到右的顺序来获取列表中的元素。

可以同时推入多个元素。

可以从列表的左端、右端或者左右两端删减任意数量的元素。

字符串的修剪操作 LTRIM 和 LRANGE 操作
 列表的弹出操作 LPOP 和 RPOP 操作 ^①
 Redis 4.0 版本

block 操作
 1 3-4
 图

3-4

命令	说明
BLPOP	BLPOP key-name [key-name ...] timeout——阻塞式弹出操作，在指定的 key 列表中按照顺序弹出元素，如果 timeout 为 0，则立即返回，否则在指定的 timeout 时间内等待。
BRPOP	BRPOP key-name [key-name ...] timeout——阻塞式弹出操作，在指定的 key 列表中按照逆序弹出元素，如果 timeout 为 0，则立即返回，否则在指定的 timeout 时间内等待。
RPOPLPUSH	RPOPLPUSH source-key dest-key——将 source-key 列表中的最后一个元素移动到 dest-key 列表的开头。
BRPOPLPUSH	BRPOPLPUSH source-key dest-key timeout——阻塞式弹出并推入操作，将 source-key 列表中的最后一个元素移动到 dest-key 列表的开头，如果在指定的 timeout 时间内没有元素可弹出，则返回空值。

第6章 Redis 3-4 BRPOPLPUSH

Redis 3-4 BLP0P

第3-4 Redis

将一个元素从一个列表移动到另一个列表，并返回被移动的元素。

当列表不包含任何元素时，阻塞弹出操作会在给定的时限内等待可弹出的元素出现，并在时限到达后返回 None（交互终端不会打印这个值）。

```
>>> conn.rpush('list', 'item1')
1
>>> conn.rpush('list', 'item2')
2
>>> conn.rpush('list2', 'item3')
1
>>> conn.brpoplpush('list2', 'list', 1)
'item3'
>>> conn.brpoplpush('list2', 'list', 1)
>>> conn.lrange('list', 0, -1)
['item3', 'item1', 'item2']
>>> conn.brpoplpush('list', 'list2', 1)
'item2'
>>> conn.blpop(['list', 'list2'], 1)
('list', 'item3')
>>> conn.blpop(['list', 'list2'], 1)
('list', 'item1')
>>> conn.blpop(['list', 'list2'], 1)
('list2', 'item2')
>>> conn.blpop(['list', 'list2'], 1)
>>>
```

将一些元素添加到两个列表里面。

弹出“list2”最右端的元素，并将被弹出的元素推入“list”的左端。

BLPOP 命令会从左到右地检查传入的列表，并对最先遇到的非空列表执行弹出操作。

Redis messaging

task queue 6

2.1 2.5

update_token() 更新 token 的函数

6.1.1 更新 token

更新 token 的函数，主要逻辑是：从 Redis 中取出 token，然后调用 update_token() 函数，将 token 更新为新的值。更新后的 token 会再次存入 Redis 中。

ePUBw.COM ePUBw.COM

本站所有资源均来自网络

3.3 集合

Redis集合数据类型包括无序集合、有序集合、带权重的有序集合、超集等。集合数据类型是Redis中非常常用的数据类型之一，可以用于存储一组无序的元素。集合数据类型支持多种操作，如添加元素、删除元素、判断元素是否存在等。集合数据类型还支持一些高级操作，如交集、并集、差集等。集合数据类型在Redis中的使用非常广泛，可以用于实现各种复杂的数据结构。

集合数据类型在Redis中的使用非常广泛，可以用于实现各种复杂的数据结构。集合数据类型支持多种操作，如添加元素、删除元素、判断元素是否存在等。集合数据类型还支持一些高级操作，如交集、并集、差集等。集合数据类型在Redis中的使用非常广泛，可以用于实现各种复杂的数据结构。

图3-5展示了集合数据类型的基本操作。

图3-5 集合数据类型的基本操作

命令	描述
SADD	SADD key-name item [item ...]——向集合中添加元素。如果元素已经存在于集合中，则不会添加。返回添加的元素个数。
SREM	SREM key-name item [item ...]——从集合中删除元素。如果元素不存在于集合中，则不会删除。返回删除的元素个数。
SISMEMBER	SISMEMBER key-name item——判断元素item是否存在于集合key-name中。返回1表示存在，0表示不存在。

명칭	설명
SCARD	SCARD key-name——해당 키의 원소 개수 반환
SMEMBERS	SMEMBERS key-name——해당 키의 원소 목록 반환
SRANDMEMBER	SRANDMEMBER key-name [count]——해당 키의 원소 중 count 개를 무작위로 반환 count가 생략되면 1을 기본값으로 사용하며, count가 0이면 빈 배열을 반환
SPOP	SPOP key-name——해당 키의 원소 하나를 제거하고 반환
SMOVE	SMOVE source-key dest-key item——source-key에서 item을 추출하여 dest-key에 추가 dest-key에 item이 이미 존재하면 1을, 그렇지 않으면 0을 반환

3-5장에서는 Redis의 집합 관련 명령어에 대해 알아보겠습니다.

3-5 Redis의 집합 관련 명령어

SADD 命令会将那些目前并不存在于集合里面的元素添加到集合里面,并返回被添加元素的数量。

获取集合包含的所有元素。

```
>>> conn.sadd('set-key', 'a', 'b', 'c')
3
>>> conn.srem('set-key', 'c', 'd')
True
>>> conn.srem('set-key', 'c', 'd')
False
>>> conn.scard('set-key')
2
>>> conn.smembers('set-key')
set(['a', 'b'])
>>> conn.smove('set-key', 'set-key2', 'a')
True
>>> conn.smove('set-key', 'set-key2', 'c')
False
>>> conn.smembers('set-key2')
set(['a'])
```

srem 函数在元素被成功移除时返回 True, 移除失败时返回 False; 注意这是 Python 客户端的一个 bug, 实际上 Redis 的 SREM 命令返回的是被移除元素的数量, 而不是布尔值。

查看集合包含的元素数量。

可以很容易地将元素从一个集合移动到另一个集合。在执行 SMOVE 命令时, 如果用户想要移动的元素不存在于第一个集合里, 那么移动操作就不会执行。

Redis 集合的添加和删除操作都是原子的, 因此可以安全地在多线程环境中使用。集合的添加和删除操作都是 O(1) 的复杂度, 而集合的成员操作是 O(N) 的复杂度。集合的遍历操作是 O(N) 的复杂度, 因此对于大型集合, 遍历操作可能会比较慢。集合的交集、并集、差集等操作都是 O(N) 的复杂度, 因此对于大型集合, 这些操作可能会比较慢。集合的排序操作是 O(N log N) 的复杂度, 因此对于大型集合, 排序操作可能会比较慢。集合的随机操作是 O(1) 的复杂度, 因此对于大型集合, 随机操作可能会比较快。

3-6 Redis 集合操作

命令	描述
SDIFF	SDIFF key-name [key-name ...]——返回两个或多个集合的差集。返回的集合包含所有不在任何指定集合中的元素。
SDIFFSTORE	SDIFFSTORE dest-key key-name [key-name ...]——将两个或多个集合的差集存储在 dest-key 中。
SINTER	SINTER key-name [key-name ...]——返回两个或多个集合的交集。返回的集合包含所有在指定集合中的元素。

命令	命令格式
SINTERSTORE	SINTERSTORE dest-key key-name [key-name ...]—— 将两个或多个集合的交集保存到dest-key中
SUNION	SUNION key-name [key-name ...]—— 计算两个或多个集合的并集
SUNIONSTORE	SUNIONSTORE dest-key key-name [key-name ...]—— 将两个或多个集合的并集保存到dest-key中

Redis 3.0 版本开始支持 SINTERSTORE、SUNION 和 SUNIONSTORE 命令。
Redis 3-6 版本支持 SINTERSTORE、SUNION 和 SUNIONSTORE 命令。

图 3-6 Redis 集合操作命令

计算出从第一个集合里面移除第二个集合包含的所有元素的结果。

```
>>> conn.sadd('skey1', 'a', 'b', 'c', 'd')
4
>>> conn.sadd('skey2', 'c', 'd', 'e', 'f')
4
>>> conn.sdiff('skey1', 'skey2')
set(['a', 'b'])
>>> conn.sinter('skey1', 'skey2')
set(['c', 'd'])
>>> conn.sunion('skey1', 'skey2')
set(['a', 'c', 'b', 'e', 'd', 'f'])
```

首先将一些元素添加到两个集合里面。

计算出同时存在于两个集合里面的所有元素。

计算出两个集合包含的所有各不相同的元素。

Python 操作 Redis 集合操作命令

Redis

[ePUBw.COM](#) [ePUBw.COM](#)

3.4 命令

图1展示了Redis命令的命名规则。Redis命令的命名规则如下：
Redis命令的命名规则如下：
命令的命名规则如下：

命令的命名规则如下：
命令的命名规则如下：
命令的命名规则如下：
命令的命名规则如下：
命令的命名规则如下：

图3-7 命令的命名规则

命令	命令格式
HMGET	HMGET key-name key [key ...]——返回存储在key中的多个字段的值
HMSET	HMSET key-name key value [key value ...]——将key中的多个字段的值设置为value
HDEL	HDEL key-name key [key ...]——删除存储在key中的多个字段的值
HLEN	HLEN key-name——返回存储在key中的字段的数量

图3-7展示了HDEL命令和HLEN命令。HMGET和HMSET命令可以一次将多个键值对添加到散列里面。Redis没有实现HMDEL命令。图3-7展示了Redis命令。

图3-7 Redis命令

使用 HMSET 命令可以一次将多个键值对添加到散列里面。

```
>>> conn.hmset('hash-key', {'k1': 'v1', 'k2': 'v2', 'k3': 'v3'})
True
>>> conn.hmget('hash-key', ['k2', 'k3'])
['v2', 'v3']
>>> conn.hlen('hash-key')
3
>>> conn.hdel('hash-key', 'k1', 'k3')
True
```

使用 HMGET 命令可以一次获取多个键的值。

HLEN命令通常用于调试一个包含非常多键值对的散列。

HDEL 命令在成功地移除了至少一个键值对时返回 True，因为 HDEL 命令已经可以同时删除多个键值对了，所以 Redis 没有实现 HMDEL 命令。

图1展示了HGET命令和HSET命令。HMGET和HMSET命令可以一次将多个键值对添加到散列里面。

图3-8 Redis命令

图3-8 Redis命令

命令	命令格式
HEXISTS	HEXISTS key-name key——返回布尔值
HKEYS	HKEYS key-name——返回键的列表

命令	命令格式
HVALS	HVALS key-name——返回所有key的值
HGETALL	HGETALL key-name——返回所有key的值
HINCRBY	HINCRBY key-name key increment——对key的值增加increment
HINCRBYFLOAT	HINCRBYFLOAT key-name key increment——对key的值增加increment

HGETALL、HKEYS、HVALS命令返回的是所有key的值，而HKEYS、HGET命令返回的是所有key的key。HGET命令返回的是所有key的key，而HGETALL命令返回的是所有key的值。

HINCRBY、HINCRBYFLOAT命令返回的是对key的值增加increment后的值。INCRBY、INCRBYFLOAT命令返回的是对key的值增加increment后的值。HINCRBY、HINCRBYFLOAT命令返回的是对key的值增加increment后的值。

3-8 Redis 命令

```
>>> conn.hmset('hash-key2', {'short':'hello', 'long':1000*'1'})
True
>>> conn.hkeys('hash-key2')
['long', 'short']
>>> conn.exists('hash-key2', 'num')
False
>>> conn.hincrby('hash-key2', 'num')
1L
>>> conn.exists('hash-key2', 'num')
True
```

在考察散列的时候，我们可以只取出散列包含的键，避免传输体积较大的值。

检查给定的键是否存在于散列中。

和字符串一样，对散列中一个尚未存在的键执行自增操作时，Redis 会将键的值当作 0 来处理。

Redis 命令 HKEYS

Redis 命令 SISMEMBER

Redis 命令 HEXISTS

Redis 命令 HINCRBY

Redis 命令

Redis 命令 ePUBw.COM

Redis 命令

3.5 数据操作

Redis 提供了对集合的操作，包括添加、删除、交集、并集、差集等。在 Redis 中，集合操作通常使用 `set` 命令。例如，添加元素可以使用 `set` 命令，删除元素可以使用 `srem` 命令。交集、并集、差集操作可以使用 `smembers`、`sunion`、`sdiff` 等命令。此外，Redis 还提供了 `scan` 命令用于遍历集合中的元素。在 Redis 中，集合操作通常使用 `set` 命令。例如，添加元素可以使用 `set` 命令，删除元素可以使用 `srem` 命令。交集、并集、差集操作可以使用 `smembers`、`sunion`、`sdiff` 等命令。此外，Redis 还提供了 `scan` 命令用于遍历集合中的元素。在 Redis 中，集合操作通常使用 `set` 命令。例如，添加元素可以使用 `set` 命令，删除元素可以使用 `srem` 命令。交集、并集、差集操作可以使用 `smembers`、`sunion`、`sdiff` 等命令。此外，Redis 还提供了 `scan` 命令用于遍历集合中的元素。

Redis 提供了对集合的操作，包括添加、删除、交集、并集、差集等。在 Redis 中，集合操作通常使用 `set` 命令。例如，添加元素可以使用 `set` 命令，删除元素可以使用 `srem` 命令。交集、并集、差集操作可以使用 `smembers`、`sunion`、`sdiff` 等命令。此外，Redis 还提供了 `scan` 命令用于遍历集合中的元素。在 Redis 中，集合操作通常使用 `set` 命令。例如，添加元素可以使用 `set` 命令，删除元素可以使用 `srem` 命令。交集、并集、差集操作可以使用 `smembers`、`sunion`、`sdiff` 等命令。此外，Redis 还提供了 `scan` 命令用于遍历集合中的元素。

图 3-9 Redis 集合操作命令

表 3-9 Redis 集合操作命令

命令	描述
ZADD	ZADD key-name score member [score member ...]——向有序集合添加元素
ZREM	ZREM key-name member [member ...]——从有序集合中删除元素
ZCARD	ZCARD key-name——返回有序集合中的元素个数

명칭	구문
ZINCRBY	ZINCRBY key-name increment member—member의 값을 increment
ZCOUNT	ZCOUNT key-name min max—min과 max 사이의 원소 수
ZRANK	ZRANK key-name member—member의 순위
ZSCORE	ZSCORE key-name member—member의 점수
ZRANGE	ZRANGE key-name start stop [WITHSCORES]—start와 stop 사이의 원소와 점수 start와 stop은 0부터 128-1까지의 정수 WITHSCORES를 지정하면 점수도 반환

이 표에서는 Redis 2.8.19 버전의 ZSET 관련 명령어를 소개한다. Redis 3.0 버전에서는 ZSET 관련 명령어가 3-9번으로 변경되었다.

3-9 Redis ZSET 관련 명령어

在 Python 客户端执行 ZADD 命令需要先输入成员、后输入分值，这跟 Redis 标准的先输入分值、后输入成员的做法正好相反。

获取指定成员的排名（排名以 0 为开始），之后可以根据这个排名来决定 ZRANGE 的访问范围。

从有序集合里面移除成员和添加成员一样容易。

```
>>> conn.zadd('zset-key', 'a', 3, 'b', 2, 'c', 1)
3
>>> conn.zcard('zset-key')
3
>>> conn.zincrby('zset-key', 'c', 3)
4.0
>>> conn.zscore('zset-key', 'b')
2.0
>>> conn.zrank('zset-key', 'c')
2
>>> conn.zcount('zset-key', 0, 3)
2L
```

取得有序集合的大小可以让我们在某些情况下知道是否需要有序集合进行修剪。

跟字符串和散列一样，有序集合的成员也可以执行自增操作。

获取单个成员的分值对于实现计数器或者排行榜之类的功能非常有用。

对于某些任务来说，统计给定分值范围内的元素数量非常有用。

```
>>> conn.zrem('zset-key', 'b')
True
>>> conn.zrange('zset-key', 0, -1, withscores=True)
[('a', 3.0), ('c', 4.0)]
```

在进行调试时，我们通常会使用 ZRANGE 取出有序集合包含的所有元素，但是在实际用例中，通常一次只会取出一小部分元素。

ZADD ZREM ZINCRBY ZSCORE ZRANGE

ZCOUNT

3-10

3-10

命令	说明
ZREVRANK	ZREVRANK key-name member——返回 member 在有序集合中的排名
ZREVRANGE	ZREVRANGE key-name start stop [WITHSCORES]——返回有序集合中指定范围的成员和分值

命令	命令格式
ZRANGEBYSCORE	ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]——返回有序集合中指定分数范围[min,max]内的元素
ZREVRANGEBYSCORE	ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]——返回有序集合中指定分数范围[min,max]内的元素，按分数降序排列
ZREMRANGEBYRANK	ZREMRANGEBYRANK key-name start stop——返回有序集合中指定排名[start,stop]范围内的元素
ZREMRANGEBYSCORE	ZREMRANGEBYSCORE key-name min max——返回有序集合中指定分数范围[min,max]内的元素
ZINTERSTORE	ZINTERSTORE dest-key key-count key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM MIN MAX]——返回两个或多个有序集合的交集，并存储在dest-key中
ZUNIONSTORE	ZUNIONSTORE dest-key key-count key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM MIN MAX]——返回两个或多个有序集合的并集，并存储在dest-key中

图3-10展示了ZINTERSTORE和ZUNIONSTORE命令的用法。ZREV*命令用于对有序集合进行反向排序。图3-10展示了ZINTERSTORE和ZUNIONSTORE命令的用法。

图3-10 ZINTERSTORE和ZUNIONSTORE命令的用法

首先创建两个有序集合。

```
>>> conn.zadd('zset-1', 'a', 1, 'b', 2, 'c', 3)
3
>>> conn.zadd('zset-2', 'b', 4, 'c', 1, 'd', 0)
3
>>> conn.zinterstore('zset-i', ['zset-1', 'zset-2'])
2L
>>> conn.zrange('zset-i', 0, -1, withscores=True)
[('c', 4.0), ('b', 6.0)]
>>> conn.zunionstore('zset-u', ['zset-1', 'zset-2'], aggregate='min')
4L
>>> conn.zrange('zset-u', 0, -1, withscores=True)
[('d', 0.0), ('a', 1.0), ('c', 1.0), ('b', 2.0)]
>>> conn.sadd('set-1', 'a', 'd')
2
>>> conn.zunionstore('zset-u2', ['zset-1', 'zset-2', 'set-1'])
4L
>>> conn.zrange('zset-u2', 0, -1, withscores=True)
[('d', 1.0), ('a', 2.0), ('c', 4.0), ('b', 6.0)]
```

ZINTERSTORE 和 ZUNIONSTORE 默认使用的聚合函数为 sum，这个函数会把各个有序集合的成员的分数都加起来。

用户可以在执行并集运算和交集运算的时候传入不同的聚合函数，共有 sum、min、max 三个聚合函数可选。

用户还可以把集合作为输入传给 ZINTERSTORE 和 ZUNIONSTORE，命令会将集合看作是成员分值全为 1 的有序集合来处理。

图3-11展示了ZINTERSTORE和ZUNIONSTORE命令的用法。图3-11展示了ZINTERSTORE和ZUNIONSTORE命令的用法。图3-11展示了ZINTERSTORE和ZUNIONSTORE命令的用法。

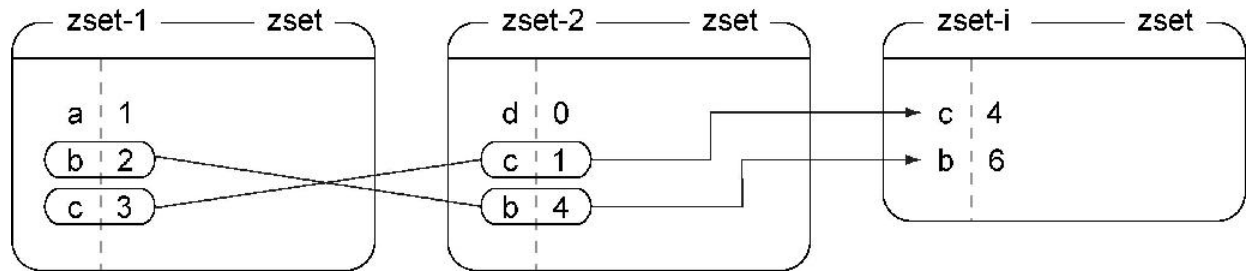


图3-1 `conn.zinterstore('zset-i', ['zset-1', 'zset-2'])` 将 zset-1 和 zset-2 的交集存储到 zset-i

图3-2 展示了 `min` 聚合函数在 ZUNIONSTORE 命令中的应用。该命令将 zset-1 和 zset-2 的并集存储到 zset-u，并使用 `min` 函数对每个元素的值进行聚合。zset-1 包含 (a, 1), (b, 2), (c, 3)，zset-2 包含 (d, 0), (c, 1), (b, 4)。zset-u 包含 (d, 0), (a, 1), (c, 1), (b, 2)。注意，zset-u 中的 (c, 1) 是 zset-1 中的 (c, 3) 和 zset-2 中的 (c, 1) 的最小值。

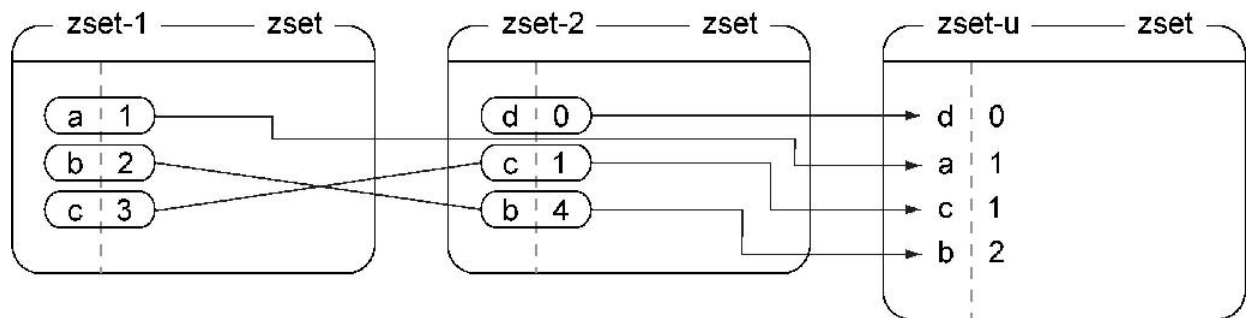


图3-2 `conn.zunionstore('zset-u', ['zset-1', 'zset-2'], aggregate='min')` 将 zset-1 和 zset-2 的并集存储到 zset-u，并使用 `min` 函数对每个元素的值进行聚合

图3-3 展示了 `ZUNIONSTORE` 命令的应用。该命令将 zset-1 和 zset-2 的并集存储到 zset-u，并使用 `min` 函数对每个元素的值进行聚合。zset-1 包含 (a, 1), (b, 2), (c, 3)，zset-2 包含 (d, 0), (c, 1), (b, 4)。zset-u 包含 (d, 0), (a, 1), (c, 1), (b, 2)。注意，zset-u 中的 (c, 1) 是 zset-1 中的 (c, 3) 和 zset-2 中的 (c, 1) 的最小值。

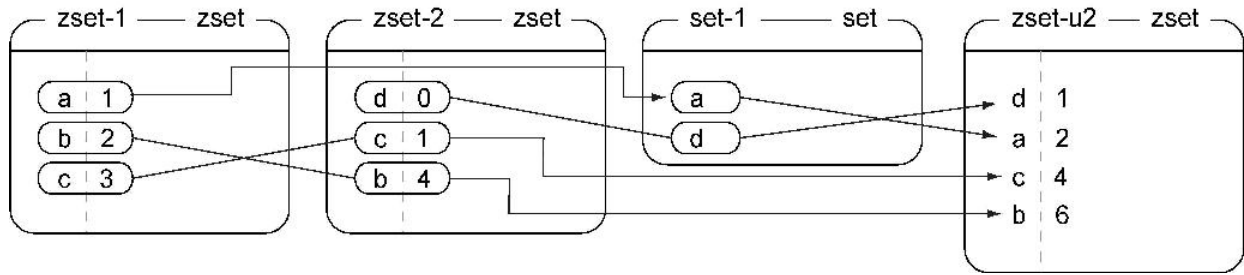


图3-3 执行conn.zunionstore('zset-u2', ['zset-1', 'zset-2', 'set-1'])
 将zset-1、zset-2和set-1合并到zset-u2中

图7展示了ZINTERSTORE和ZUNIONSTORE命令的用法。ZINTERSTORE命令用于计算多个集合的交集，而ZUNIONSTORE命令用于计算多个集合的并集。图中还展示了WEIGHTS参数的使用，用于指定每个集合的权重。

Redis的发布/订阅功能（publish/subscribe）允许用户将消息发布到指定的频道，其他订阅了该频道的客户端可以接收到消息。Redis的发布/订阅功能是基于内存的，不支持持久化。

图1展示了ePUBw.COM网站的界面。ePUBw.COM是一个提供电子书下载的网站，用户可以在该网站上搜索和下载各种格式的电子书。

图2展示了ePUBw.COM网站的搜索界面。用户可以在该界面上输入关键词进行搜索，网站会返回相关的搜索结果。

3.6

```

    pub/subListener
    channelPublisher
    binary
string message

```

Redis

3-11 Redis 5

3-11 Redis

コマンド	メッセージ
SUBSCRIBE	SUBSCRIBE channel [channel ...]—チャンネル登録
UNSUBSCRIBE	UNSUBSCRIBE [channel [channel ...]]—チャンネル登録解除 チャンネル登録解除のメッセージは、チャンネル登録解除のメッセージを受信したクライアントにのみ送信される。
PUBLISH	PUBLISH channel message—メッセージ送信

命令	命令格式
PSubscribe	PSubscribe pattern [pattern ...]——阻塞式订阅， 返回消息
PUnsubscribe	PUnsubscribe [pattern [pattern ...]]——阻塞式 取消订阅，返回消息

Redis的PUBLISH和SUBSCRIBE命令在Python中可以通过redis-py库实现。在3.11版本中，redis-py库提供了一个helper thread来辅助PUBLISH和SUBSCRIBE操作。

图3-11 Redis的PUBLISH和SUBSCRIBE命令

```

>>> def publisher(n):
...     time.sleep(1)
...     for i in xrange(n):
...         conn.publish('channel', i)
...         time.sleep(1)
>>> def run_pubsub():
...     threading.Thread(target=publisher, args=(3,)).start()
...     pubsub = conn.pubsub()
...     pubsub.subscribe(['channel'])
...     count = 0
...     for item in pubsub.listen():
...         print item
...         count += 1
...         if count == 4:
...             pubsub.unsubscribe()
...             if count == 5:
...                 break
...
>>> run_pubsub()
{'pattern': None, 'type': 'subscribe', 'channel': 'channel', 'data': 1L}
{'pattern': None, 'type': 'message', 'channel': 'channel', 'data': '0'}
{'pattern': None, 'type': 'message', 'channel': 'channel', 'data': '1'}
{'pattern': None, 'type': 'message', 'channel': 'channel', 'data': '2'}
{'pattern': None, 'type': 'unsubscribe', 'channel': 'channel', 'data': 0L}

```

函数在刚开始执行时会先休眠，让订阅者有足够的时间来连接服务器并监听消息。

启动发送者线程，并让它发送三条消息。

在发布消息之后进行短暂的休眠，让消息可以一条接一条地出现。

创建发布与订阅对象，并让它订阅给定的频道。

通过遍历函数 pubsub.listen() 的执行结果来监听订阅消息。

打印接收到的每条消息。

在接收到一条订阅反馈消息和三条发布者发送的消息之后，执行退订操作，停止监听新消息。

客户端在接收到退订反馈消息之后就不再接收消息。

实际运行函数并观察它们的行为。

在刚开始订阅一个频道的时候，客户端会接收到一条关于被订阅频道的反馈消息。

在退订频道时，客户端会接收到一条反馈消息，告知被退订的是哪个频道，以及客户端目前仍在订阅的频道数量。

这些结构就是我们在遍历 pubsub.listen() 函数时得到的元素。

Redis 8.5

Redis Redis

Redis

Redis Redis

Redis client-output-

buffer-limit pubsub 8

Python Redis connection pool 4

Redis PUBLISH SUBSCRIBE 6

Redis 8.5 client-output-buffer-limit pubsub

Redis 5

ePUBw.COM

3.7 排序

Redis 5.0 版本开始支持 SORT 命令，用于对 Redis 中的列表、集合、有序集合、哈希表等进行排序。该命令支持多种排序方式，包括按值排序、按键排序、按哈希值排序等。此外，还支持对排序结果进行限制（LIMIT）和存储到目标键（STORE）。

Redis 5.0 版本开始支持 SORT 命令，用于对 Redis 中的列表、集合、有序集合、哈希表等进行排序。该命令支持多种排序方式，包括按值排序、按键排序、按哈希值排序等。此外，还支持对排序结果进行限制（LIMIT）和存储到目标键（STORE）。

3.7.1 排序

Redis 5.0 版本开始支持 SORT 命令，用于对 Redis 中的列表、集合、有序集合、哈希表等进行排序。该命令支持多种排序方式，包括按值排序、按键排序、按哈希值排序等。此外，还支持对排序结果进行限制（LIMIT）和存储到目标键（STORE）。

Redis 5.0 版本开始支持 SORT 命令，用于对 Redis 中的列表、集合、有序集合、哈希表等进行排序。该命令支持多种排序方式，包括按值排序、按键排序、按哈希值排序等。此外，还支持对排序结果进行限制（LIMIT）和存储到目标键（STORE）。

图 3-12 SORT 命令

命令	命令格式
<code>SORT</code>	<code>SORT source-key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC DESC] [ALPHA] [STORE dest-key]</code> ——对 source-key 中的元素进行排序，并按指定的方式返回结果。

对 SORT 列表进行排序。在列表 '110' 和 '12' 中，数字 110 比 12 大，因此 110 排在 12 之前。这导致列表的顺序与数字大小相反。

在图 3-12 中，我们使用 SORT 列表进行排序。在图 3-12 中，我们使用 SORT 列表进行排序。

图 3-12 对 SORT 列表进行排序

```
>>> conn.rpush('sort-input', 23, 15, 110, 7)
```

```
4
```

```
>>> conn.sort('sort-input')
```

```
['7', '15', '23', '110']
```

根据数字大小对元素进行排序。

首先将一些元素添加到列表里面。

```
>>> conn.sort('sort-input', alpha=True)
```

```
['110', '15', '23', '7']
```

根据字母表顺序对元素进行排序。

```
>>> conn.hset('d-7', 'field', 5)
```

```
1L
```

```
>>> conn.hset('d-15', 'field', 1)
```

```
1L
```

```
>>> conn.hset('d-23', 'field', 9)
```

```
1L
```

```
>>> conn.hset('d-110', 'field', 3)
```

```
1L
```

```
>>> conn.sort('sort-input', by='d-*->field')
```

```
['15', '110', '7', '23']
```

添加一些用于执行排序操作和获取操作的附加数据。

将散列的域 (field) 用作权重，对 sort-input 列表进行排序。

```
>>> conn.sort('sort-input', by='d-*->field', get='d-*->field')
```

```
['1', '3', '5', '9']
```

获取外部数据，并将它们用作命令的返回值，而不是返回被排序的数据。

SORT 列表。在图 3-12 中，我们使用 alpha 列表进行排序。

redis 7 redis sort redis
redis sort redis sort

redis sort redis 3 redis
redis

3.7.2 redis

redis redis redis
redis
ZUNIONSTORE redis
redis 5 interruption
WATCH MULTI EXEC UNWATCH DISCARD

redis redis MULTI EXEC
WATCH MULTI EXEC UNWATCH 4.4
MULTI EXEC WATCH UNWATCH

redis

redis basic transaction MULTI EXEC
redis
rollback redis MULTI EXEC
redis redis

Redis 的 MULTI 命令可以开启事务，EXEC 命令可以执行事务。Redis 的 MULTI 命令和 Redis 的 EXEC 命令必须在同一客户端中执行。Redis 的 Python 客户端提供了 pipeline 方法，可以批量执行命令。Redis 的 pipeline 方法可以批量执行命令，提高了性能。Redis 的 pipeline 方法可以批量执行命令，提高了性能。Redis 的 pipeline 方法可以批量执行命令，提高了性能。

PUBLISH 和 SUBSCRIBE 命令可以用于发布和订阅消息。Redis 的 PUBLISH 命令可以用于发布消息，Redis 的 SUBSCRIBE 命令可以用于订阅消息。Redis 的 PUBLISH 命令可以用于发布消息，Redis 的 SUBSCRIBE 命令可以用于订阅消息。Redis 的 PUBLISH 命令可以用于发布消息，Redis 的 SUBSCRIBE 命令可以用于订阅消息。

图 3-13 使用 Redis 的 pipeline 方法

```
>>> def notrans():
...     print conn.incr('notrans:')
...     time.sleep(.1)
...     conn.incr('notrans:', -1)
...
>>> if 1:
...     for i in xrange(3):
...         threading.Thread(target=notrans).start()
...         time.sleep(.5)
...
1
2
3
```

等待 100 毫秒。

对 'notrans:' 计数器执行自增操作并打印操作的执行结果。

对 'notrans:' 计数器执行自减操作。

启动 3 个线程来执行没有被事务包裹的自增、休眠和自减操作。

等待 500 毫秒，让操作有足够的时间完成。

因为没有使用事务，所以 3 个线程执行的各个命令将互相交错，使得计数器的值持续地增大。

Redis 的 pipeline 方法可以批量执行命令，提高了性能。Redis 的 pipeline 方法可以批量执行命令，提高了性能。Redis 的 pipeline 方法可以批量执行命令，提高了性能。

3-14

3-14

创建一个事务型
(transactional) 流
水线对象。

把针对'trans:'计
数器的自减操作放
入队列。

执行被事务包裹的命
令，并打印自增操作的
执行结果。

```
>>> def trans():
...     pipeline = conn.pipeline()
...     pipeline.incr('trans:')
...     time.sleep(.1)
...     pipeline.incr('trans:', -1)
...     print pipeline.execute()[0]
>>> if 1:
...     for i in xrange(3):
...         threading.Thread(target=trans).start()
...         time.sleep(.5)
...
1
1
1
```

把针对'trans:'计数器
的自增操作放入队列。

等待 100 ms。

启动 3 个线程来执行被事务包裹的
自增、休眠和自减 3 个操作。

等待 500 ms，
让操作有足够
的时间完成。

因为每组自增、休眠和自减操作都在事务
里面执行，所以命令之间不会互相交错，
因此所有事务的执行结果都是 1。

Redis EXEC
MULTI EXEC

4.4

3-13 MULTI EXEC

1 article_vote() bug

bug

`article_vote()` 函数中 `bug` 函数中 `1` 个 `post_article()` 函数中 `6.2.5` 个

函数中

Redis 函数中 `4.4` 个 `4.6` 个
Redis 函数中 `1` 个 `get_articles()` 函数中 `Redis` 函数中
`26` 个 `get_articles()` 函数中 `26` 个 `2` 个

Redis 函数中 `Redis` 函数中 `Redis` 函数中

3.7.3 函数中

Redis 函数中 `DEL` 函数中
Redis 函数中 `expiration` 函数中
`timeout` 函数中 `“time to live”` 函数中
`expire` 函数中 Redis 函数中

6.2

7.1 7.2
container

Redis

Redis

3-13 Redis

3-13 Redis

命令	命令格式
PERSIST	PERSIST key-name——永久保存
TTL	TTL key-name——设置过期时间
EXPIRE	EXPIRE key-name seconds——设置过期时间
EXPIREAT	EXPIREAT key-name timestamp——设置过期时间UNIX 时间戳

命令	命令格式
PTTL	PTTL key-name——返回该键的过期时间，以秒为单位。Redis 2.6 之前返回的是以秒为单位的过期时间。
PEXPIRE	PEXPIRE key-name milliseconds——返回该键的过期时间，以毫秒为单位。Redis 2.6 之前返回的是以秒为单位的过期时间。
PEXPIREAT	PEXPIREAT key-name timestamp-milliseconds——返回该键的过期时间，以毫秒为单位。Redis 2.6 之前返回的是以秒为单位的过期时间。

图 3-15 Redis 过期时间设置命令

图 3-15 Redis 过期时间设置命令

```
>>> conn.set('key', 'value')
True
>>> conn.get('key')
'value'
```

设置一个简单的字符串值，
作为过期时间的设置对象。

```
>>> conn.expire('key', 2)
True
>>> time.sleep(2)
>>> conn.get('key')
>>> conn.set('key', 'value2')
True
```

如果我们为键设置了过期时间，并
在键过期后尝试获取键的值，那么
就会发现键已经被删除了。

```
>>> conn.expire('key', 100); conn.ttl('key')
True
100
```

查看键距离过期
还有多长时间。

图 3-15 Redis 过期时间设置命令

2.1 2.2 2.5 购物车 ID

购物车 ID

Redis

ID update_token()

add_to_cart() ID

ID

ePUBw.COM ePUBw.COM

3.8 消息

Redis 支持发布订阅消息功能，主要命令有 PUBLISH、SUBSCRIBE、SORT、MULTI、EXEC 等。

Redis 消息功能支持发布订阅消息，支持 70 多种消息类型，详情请见 <http://redis.io/commands>。

Redis 消息功能支持发布订阅消息，支持 1 个发布者和 2 个订阅者，支持 1 个发布者和 2 个订阅者，支持 1 个发布者和 2 个订阅者。

Redis 消息功能支持发布订阅消息，支持 1 个发布者和 2 个订阅者，支持 1 个发布者和 2 个订阅者。

① Redis 消息功能支持发布订阅消息，支持 1 个发布者和 2 个订阅者，支持 1 个发布者和 2 个订阅者。

② Redis 消息功能支持发布订阅消息，支持 1 个发布者和 2 个订阅者，支持 1 个发布者和 2 个订阅者。

③ Redis 消息功能支持发布订阅消息，支持 1 个发布者和 2 个订阅者，支持 1 个发布者和 2 个订阅者。

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

4

--	--	--	--	--	--

- Redis
- Redis pipeline
- Redis non-transactional pipeline

`Redis`

`Redis`

`Redis`

[illegible]

Redis

RedisRedis

ePUBw.COM ePUBw.COM

4.1 持久化

Redis持久化是指将内存中的数据集持久化的过程。

Redis持久化分为两种：快照持久化（snapshotting）和追加只读文件持久化（append-only file，AOF）。

快照持久化是指将内存中的数据集定期地写入到磁盘中的快照文件中。快照文件是Redis持久化的主要方式，也是Redis持久化的默认方式。快照文件是Redis持久化的主要方式，也是Redis持久化的默认方式。

Redis持久化是指将内存中的数据集持久化的过程。Redis持久化分为两种：快照持久化（snapshotting）和追加只读文件持久化（append-only file，AOF）。Redis持久化是指将内存中的数据集持久化的过程。Redis持久化分为两种：快照持久化（snapshotting）和追加只读文件持久化（append-only file，AOF）。Redis持久化是指将内存中的数据集持久化的过程。Redis持久化分为两种：快照持久化（snapshotting）和追加只读文件持久化（append-only file，AOF）。

Redis持久化是指将内存中的数据集持久化的过程。Redis持久化分为两种：快照持久化（snapshotting）和追加只读文件持久化（append-only file，AOF）。Redis持久化是指将内存中的数据集持久化的过程。Redis持久化分为两种：快照持久化（snapshotting）和追加只读文件持久化（append-only file，AOF）。

4-1 Redis持久化

```

save 60 1000
stop-writes-on-bgsave-error no
rdbcompression yes
dbfilename dump.rdb

appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

dir ./

```

快照持久化选项。

AOF 持久化选项。

共享选项，这个选项决定了快照文件和 AOF 文件的保存位置。

4-1 持久化选项

持久化选项

AOF subsystem Redis AOF

AOF compaction AOF

4.1.1 持久化

Redis 持久化

持久化选项

持久化选项

dbfilename dir

Redis Redis

Redis 10GB 2:35

3:06 Redis 3:08

00:35 00:00 00:00 00:00 3:06 00:00 3:08 00:00 00:00 00:00 Redis 00:00 00:00 00:00 Redis 00:00 2:35 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 Redis 00:00 35 00:00 00:00

00:00 00:00 00:00 00:00

- 00:00 00:00 Redis 00:00 BGSAVE 00:00 00:00 00:00 00:00 BGSAVE 00:00 00:00 00:00 00:00 Windows 00:00 Redis 00:00 fork^① 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00
- 00:00 00:00 Redis 00:00 SAVE 00:00 00:00 00:00 00:00 SAVE 00:00 Redis 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 SAVE 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 BGSAVE 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00
- 00:00 00:00 save 00:00 00:00 save 60 10000 00:00 Redis 00:00 00:00 00:00 00:00 “60 00:00 10 000 00:00” 00:00 00:00 00:00 Redis 00:00 00:00 00:00 BGSAVE 00:00 00:00 00:00 00:00 save 00:00 00:00 00:00 00:00 save 00:00 00:00 00:00 00:00 00:00 Redis 00:00 00:00 BGSAVE 00:00
- 00:00 Redis 00:00 SHUTDOWN 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 TERM 00:00 00:00 00:00 SAVE 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00
- 00:00 Redis 00:00 00:00 00:00 Redis 00:00 00:00 00:00 SYNC 00:00 00:00 00:00 00:00 00:00 00:00 00:00 00:00 BGSAVE 00:00 00:00 00:00 00:00 00:00 BGSAVE 00:00 00:00 00:00 00:00 BGSAVE 00:00 00:00 00:00 00:00 4.2 00:00

Redis 4.0 版本开始，Redis 支持了两种持久化策略：RDB 和 AOF。RDB 是 Redis 的默认持久化策略，而 AOF 则是 Redis 4.0 版本新增的持久化策略。AOF 的全称是 Append Only File，即只追加文件。AOF 的持久化策略是将 Redis 中的每一个写操作都追加到 AOF 文件中，当 Redis 重启时，可以通过 AOF 文件来恢复 Redis 中的数据。

1. Redis 持久化策略

Redis 持久化策略分为两种：RDB 和 AOF。RDB 是 Redis 的默认持久化策略，而 AOF 则是 Redis 4.0 版本新增的持久化策略。RDB 的持久化策略是将 Redis 中的数据进行快照，并将快照保存到磁盘上。AOF 的持久化策略是将 Redis 中的每一个写操作都追加到 AOF 文件中，当 Redis 重启时，可以通过 AOF 文件来恢复 Redis 中的数据。

Redis 持久化策略的配置文件是 `redis.conf`。在 `redis.conf` 文件中，可以通过配置 `save` 和 `appendonly` 来启用或禁用持久化策略。例如，配置 `save 900 1` 表示每隔 900 秒将 Redis 中的数据快照保存到磁盘上。配置 `appendonly yes` 表示启用 AOF 持久化策略。

Redis 持久化策略的配置文件是 `redis.conf`。在 `redis.conf` 文件中，可以通过配置 `save` 和 `appendonly` 来启用或禁用持久化策略。例如，配置 `save 900 1` 表示每隔 900 秒将 Redis 中的数据快照保存到磁盘上。配置 `appendonly yes` 表示启用 AOF 持久化策略。

2. Redis 持久化策略的配置文件

Redis 持久化策略的配置文件是 `redis.conf`。在 `redis.conf` 文件中，可以通过配置 `save` 和 `appendonly` 来启用或禁用持久化策略。例如，配置 `save 900 1` 表示每隔 900 秒将 Redis 中的数据快照保存到磁盘上。配置 `appendonly yes` 表示启用 AOF 持久化策略。

4-2 Redis 的 pipeline 方法

4-2 process_logs() 函数与 Redis

获取文件当前的
处理进度。

```
def process_logs(conn, path, callback):
    current_file, offset = conn.mget(
        'progress:file', 'progress:position')
    pipe = conn.pipeline()
```

日志处理函数接受的其
中一个参数为回调函数，
这个回调函数接受一个
Redis 连接和一个日志行
作为参数，并通过调用流
水线对象的方法来执行
Redis 命令。

通过使用闭包
(closure) 来减
少重复代码。

```
def update_progress():
    pipe.mset({
        'progress:file': fname,
        'progress:position': offset
    })
    pipe.execute()
```

对正在处理的日志
文件的名字和偏移
量进行更新。

这个语句负责执行实
际的日志更新操作，并
将日志文件的名字和
目前的处理进度记录
到 Redis 里面。

```
for fname in sorted(os.listdir(path)):
    if fname < current_file:
        continue
```

有序地遍历各个
日志文件。

在接着处理一个因为
系统崩溃而未能完成
处理的日志文件时，略
过已处理的内容。

```
inp = open(os.path.join(path, fname), 'rb')
if fname == current_file:
    inp.seek(int(offset), 10)
else:
    offset = 0
current_file = None
```

略过所有已处理的
日志文件。

枚举函数遍历一个由文件
行组成的序列，并返回任
意多个二元组，每个二元组
包含了行号 lno 和行数据
line，其中行号从 0 开始。

```
for lno, line in enumerate(inp):
    callback(pipe, line)
    offset += int(offset) + len(line)
    if not (lno+1) % 1000:
        update_progress()
    update_progress()
inp.close()
```

处理日志行。

更新已处理内容
的偏移量。

每当处理完 1000 个日志行或者
处理完整个日志文件的时候，都
更新一次文件的处理进度。

Redis 的 pipeline 方法

3

□□□□

3□□□

□Redis□□□□□□□□GB□□□□□□□□□□□□□□Redis□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Redis□□□□□
□□□BGSAVE□□□□□□□□□□□□□□□□Redis□□□□□□□□□□GB□□
□□□□□□□□□□□□Redis□□□□□□□virtual machine□□□□□□□
BGSAVE□□□□□□□□□□□□□□□□□□□□□□□□□□virtual
memory□□□□□□Redis□□□□□□□□□□□□□□

□□BGSAVE□□□□□□□□□□□□Redis□□□□□□□□□□□□□□VMWare
□□□□KVM□□□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□
10□20□□□□□□□Xen□□□□□□□□□□□□□□Redis□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□Redis□□□□□20 GB□□□□□
□□□□□□□□□□BGSAVE□□□□□□□□□□Redis□□200□400□□□□□□□□□□
Xen□□□□□□□□EC2□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Redis□□4□6□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
BGSAVE□□SAVE□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□BGSAVE□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
SAVE□□□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Redis 的持久化策略有 RDB 和 AOF 两种。RDB 是 Redis 的默认持久化策略，它通过快照的方式将 Redis 中的数据持久化到磁盘。AOF 则是通过记录 Redis 的写操作来持久化数据。

Redis 的持久化策略选择主要取决于数据量和性能要求。对于数据量较小的 Redis，RDB 是更好的选择，因为它对性能的影响较小。对于数据量较大的 Redis，AOF 是更好的选择，因为它可以保证数据的完整性。Redis 的持久化策略还可以通过配置来调整，例如设置 RDB 的快照间隔和 AOF 的同步策略。

Redis 的持久化策略还可以通过配置来调整，例如设置 RDB 的快照间隔和 AOF 的同步策略。Redis 的持久化策略还可以通过配置来调整，例如设置 RDB 的快照间隔和 AOF 的同步策略。

4.1.2 AOF 持久化

AOF 持久化是 Redis 的一种持久化策略，它通过记录 Redis 的写操作来持久化数据。AOF 持久化的优点是数据完整性高，缺点是性能开销较大。Redis 的持久化策略可以通过配置来调整，例如设置 AOF 的同步策略。

Redis 的持久化策略还可以通过配置来调整，例如设置 RDB 的快照间隔和 AOF 的同步策略。Redis 的持久化策略还可以通过配置来调整，例如设置 RDB 的快照间隔和 AOF 的同步策略。

appendfsync always sync Redis data to disk every time a write command is executed. This is the safest option but it can be slow.

4-1 appendfsync options

Option	Description
always	Redis syncs data to disk every time a write command is executed.
everysec	Redis syncs data to disk every second.
no	Redis does not sync data to disk.

appendfsync always syncs Redis data to disk every time a write command is executed. This is the safest option but it can be slow. Redis uses a spinning disk (HDD) which has a latency of about 200 microseconds. Solid-state drive (SSD) has a latency of about 10 microseconds.

appendfsync always syncs Redis data to disk every time a write command is executed. This is the safest option but it can be slow. Redis uses a spinning disk (HDD) which has a latency of about 200 microseconds. Solid-state drive (SSD) has a latency of about 10 microseconds. write amplification is a measure of the amount of data written to disk relative to the amount of data requested by the client.

appendfsync everysec syncs Redis data to disk every second. This is a good compromise between safety and performance. Redis uses a spinning disk (HDD) which has a latency of about 200 microseconds. Solid-state drive (SSD) has a latency of about 10 microseconds.


```

    AOF Redis BGREWRITEAOF
rewrite AOF BGSAVE Redis
AOF AOF
AOF AOF
AOF AOF
AOF AOF
AOF AOF
GB AOF hang

```

```

save BGSAVE AOF
auto-aof-rewrite-percentage auto-aof-rewrite-min-
size BGREWRITEAOF Redis auto-
aof-rewrite-percentage 100 auto-aof-rewrite-min-
size 64mb AOF AOF 64 MB AOF
Redis 100% Redis BGREWRITEAOF
AOF auto-aof-rewrite-
percentage 100 Redis AOF
Redis
```

AOF

AOF

4.2 Redis

Redis replication architecture. The architecture consists of one master Redis instance and multiple slave Redis instances. The master instance is responsible for all write operations, and the slave instances are responsible for read operations. The master instance replicates data to the slave instances. The slave instances can also replicate data to other slave instances, forming a replication chain.

Redis replication is a synchronous process. The master instance sends a replication stream to the slave instances. The slave instances receive the stream and apply the commands to their local database. The slave instances can also replicate data to other slave instances, forming a replication chain. The master instance can have up to 10 slave instances. The slave instances can have up to 100 slave instances.

SUNIONSTORE command. The **SUNIONSTORE** command is used to create a new Redis instance. The command takes the following parameters: **SUNIONSTORE** **2.4 GHz** **10 000** **SUNIONSTORE** **20 000** **Redis** **initial copy of the data**

The **SUNIONSTORE** command is used to create a new Redis instance. The command takes the following parameters: **SUNIONSTORE** **2.4 GHz** **10 000** **SUNIONSTORE** **20 000** **Redis** **initial copy of the data**

Redis Redis

4.2.1 Redis

4.1.1 BGSAVE

4-1 dir

dbfilename Redis

writable

slaveof

Redis slaveof host port

Redis IP Redis

SLAVEOF no one

SLAVEOF host port

Redis Redis

4.2.2 Redis

4-2

Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-2 中 Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-2 中

Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-2 中 Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-2 中

Redis 的 master-master replication 命令和 Redis 的 SLAVEOF 命令在 4-2 中 Redis 的 master-master replication 命令和 Redis 的 SLAVEOF 命令在 4-2 中

Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-3 中 Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-3 中

表 4-3 Redis 的 SLAVEOF 命令和 Redis 的 AOF 命令在 4-3 中

Redis 的 SLAVEOF 命令	Redis 的 AOF 命令
Redis 的 SLAVEOF 命令	Redis 的 AOF 命令
Redis 的 SLAVEOF 命令	Redis 的 AOF 命令

Redis 的持久化功能，可以将内存中的数据保存到磁盘上，防止数据丢失。Redis 提供了两种持久化方式：RDB 和 AOF。RDB 是 Redis 的默认持久化方式，它会将内存中的数据快照保存到磁盘上。AOF 则是将 Redis 的写操作记录到日志中，通过重放日志来恢复数据。Redis 还支持混合持久化，即同时使用 RDB 和 AOF。

4.2.3 主从复制

Redis 的主从复制功能，可以将一个 Redis 实例（主节点）的数据复制到多个 Redis 实例（从节点）中。主从复制可以提高 Redis 的可用性和性能。主节点负责处理所有的写操作，而从节点则负责处理所有的读操作。主从复制的配置非常简单，只需要在主节点的配置文件中设置从节点的 IP 地址和端口即可。

Redis 的主从复制分为全量复制和增量复制。全量复制是在主从节点初次建立连接时进行的，会将主节点的所有数据复制到从节点。增量复制则是主从节点建立连接后，主节点会将新的写操作记录到日志中，从节点通过重放日志来同步数据。Redis 还支持主从链式复制，即一个从节点可以作为另一个从节点的主节点。

Redis 的主从复制功能，可以用于提高 Redis 的可用性和性能。主节点负责处理所有的写操作，而从节点则负责处理所有的读操作。主从复制的配置非常简单，只需要在主节点的配置文件中设置从节点的 IP 地址和端口即可。Redis 还支持主从链式复制，即一个从节点可以作为另一个从节点的主节点。主从复制的配置非常简单，只需要在主节点的配置文件中设置从节点的 IP 地址和端口即可。

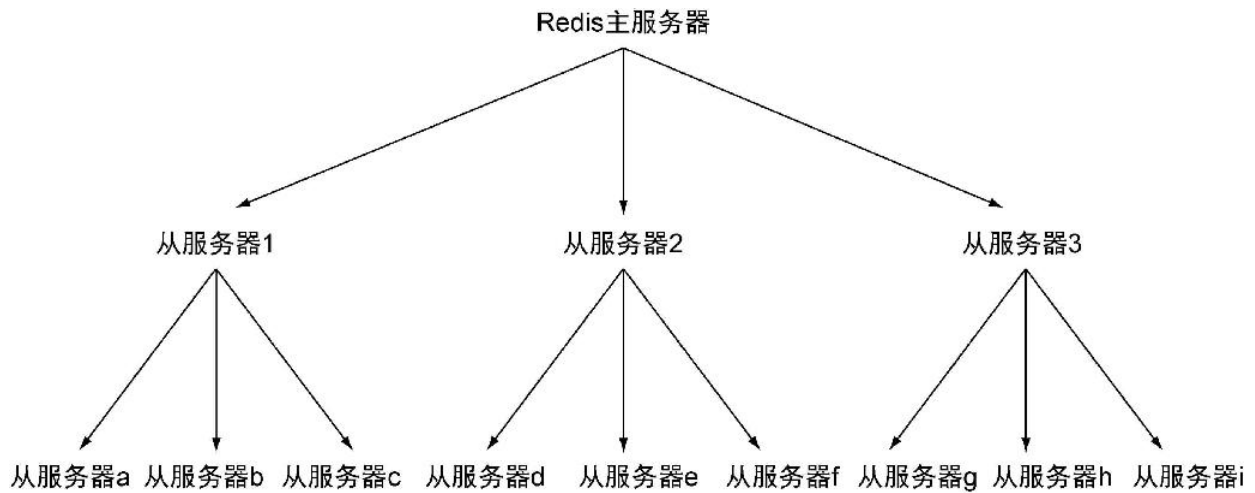


图4-1 Redis master/slave replica tree 3 9

图4-1 Redis master/slave replica tree 3 9

Redis possible reasonable

4.1.2 AOF

1 AOF

appendonly yes appendfsync everysec

4.2.4

INFO Redis INFO

wait_for_sync()
sync wait ZSET
1
1

AOF Redis

ePUBw.COM ePUBw.COM

4.3 ☐☐☐☐☐☐

[illegible]

4.3.1 五五五五五AOF

```

redis-check-aof redis-check-dump redis-check-aof redis-check-dump
redis-check-aof redis-check-dump redis-check-aof redis-check-dump
redis-check-aof redis-check-dump redis-check-aof redis-check-dump
redis-check-aof redis-check-dump redis-check-aof redis-check-dump

```

```
$ redis-check-aof
Usage: redis-check-aof [--fix] <file.aof>
$ redis-check-dump
Usage: redis-check-dump <dump.rdb>
$
```

redis-check-aof --fix AOF
AOF AOF
AOF

SHA1
SHA256 Linux Unix sha1sum
sha256sum

checksum **hash** 2.6 Redis
CRC64 CRC
SHA arbitrary error
64 subset
CRC64 SHA1 SHA256

Redis

4.3.2

Redis Redis

在机器 B 上，使用 Redis 客户端执行以下命令：

在机器 A 上，使用 Redis 客户端执行以下命令：

在机器 B 上，使用 Redis 客户端执行以下命令：

4-4 在机器 B 上执行命令

图 4-4 在机器 B 上执行命令

```
user@vpn-master ~:$ ssh root@machine-b.vpn
Last login: Wed Mar 28 15:21:06 2012 from ...
root@machine-b ~:$ redis-cli
redis 127.0.0.1:6379> SAVE
OK
redis 127.0.0.1:6379> QUIT
root@machine-b ~:$ scp \
> /var/local/redis/dump.rdb machine-c.vpn:/var/local/redis/
dump.rdb
100% 525MB 8.1MB/s 01:05
root@machine-b ~:$ ssh machine-c.vpn
Last login: Tue Mar 27 12:42:31 2012 from ...
root@machine-c ~:$ sudo /etc/init.d/redis-server start
Starting Redis server...
root@machine-c ~:$ exit
root@machine-b ~:$ redis-cli
redis 127.0.0.1:6379> SLAVEOF machine-c.vpn 6379
OK
redis 127.0.0.1:6379> QUIT
root@machine-b ~:$ exit
user@vpn-master ~:$
```

通过 VPN 网络连接机器 B。

执行 SAVE 命令，并在命令完成之后，使用 QUIT 命令退出客户端。

启动命令行 Redis 客户端来执行几个简单的操作。

将快照文件发送至新的主服务器——机器 C。

连接新的主服务器并启动 Redis。

告知机器 B 的 Redis，让它将机器 C 用作新的主服务器。

图 4-4 在机器 B 上执行命令

在机器 B 上，使用 Redis 客户端执行以下命令：

SLAVEOF

turn
Redis
Redis

Redis Sentinel Redis Sentinel
failover 10 Redis
Sentinel

ePUBw.COM ePUBw.COM

4.4 Redis

Redis 是一个开源的分布式数据库，它支持多种数据类型，如字符串、哈希、列表、集合等。Redis 的特点包括高性能、高可用、持久化等。Redis 可以用于缓存、消息队列、分布式锁等场景。

Redis 支持事务，事务是指一组 Redis 命令，这些命令要么都执行，要么都不执行。Redis 的事务命令包括 `BEGIN`、`EXEC`、`COMMIT` 和 `ROLLBACK`。Redis 的事务是异步执行的，这意味着事务中的命令可能会在事务开始之前或之后执行。

Redis 3.7.2 版本引入了 `MULTI` 命令，用于开始一个事务。在 `MULTI` 命令之后，可以执行一系列 Redis 命令，最后通过 `EXEC` 命令提交事务。如果事务中的任何一个命令执行失败，整个事务都会失败，并返回 `EXEC ABORT`。Redis 3.7.2 还引入了 `WATCH` 命令，用于监视一个或多个键，如果这些键被其他客户端修改，那么 `EXEC` 命令就会失败。Redis 3.7.2 还支持 `two-phase commit`，这是一种两阶段提交协议，用于保证分布式事务的原子性。

Redis 还支持 `EXEC` 命令，用于执行一个事务。在 `EXEC` 命令之前，必须先使用 `MULTI` 命令开始一个事务。Redis 还支持 `WATCH` 命令，用于监视一个或多个键，如果这些键被其他客户端修改，那么 `EXEC` 命令就会失败。

Redis 的 pipeline 功能，可以批量发送命令，Redis 会一次性执行所有命令，并返回结果。这大大减少了网络往返次数，提高了性能。

在 Fake Game 游戏中，YouTwitFace 用户需要登录。登录时，系统会检查用户名和密码是否正确。如果正确，系统会将用户信息存储在 Redis 中，并返回登录成功的信息。

4.4.1 用户登录流程

图 4-2 展示了用户登录流程。用户输入用户名和密码，系统会检查 Redis 中的用户信息。如果信息正确，系统会将用户信息存储在 Redis 中，并返回登录成功的信息。

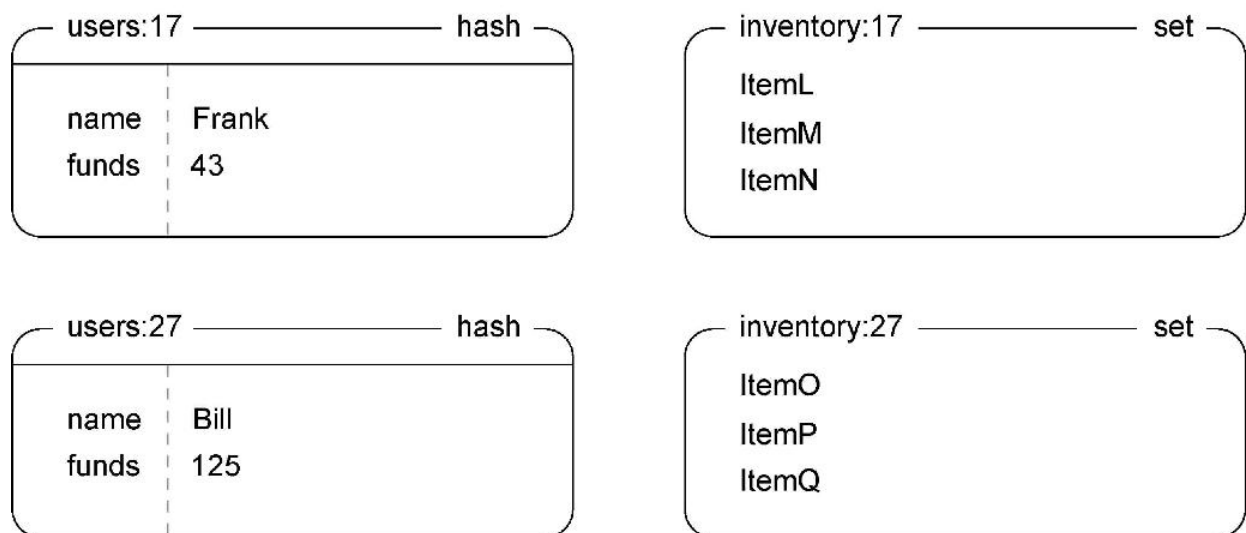


图 4-2 用户登录流程：Frank 的 funds 为 43，库存包含 ItemL、ItemM、ItemN、ItemO、ItemP 和 ItemQ。

在 Redis 中，我们使用 ZSET 来存储正在销售的商品及其价格。
 每个商品在 ZSET 中的成员名称格式为 `ItemID.所有者ID`，其权重（score）即为该商品的当前售价。
 例如，商品 `ItemA` 由所有者 `4` 销售，当前售价为 35，其在 ZSET 中的成员名称为 `ItemA.4`，权重为 35。

在 Redis 中，我们使用 ZSET 来存储正在销售的商品及其价格。
 每个商品在 ZSET 中的成员名称格式为 `ItemID.所有者ID`，其权重（score）即为该商品的当前售价。
 例如，商品 `ItemA` 由所有者 `4` 销售，当前售价为 35，其在 ZSET 中的成员名称为 `ItemA.4`，权重为 35。

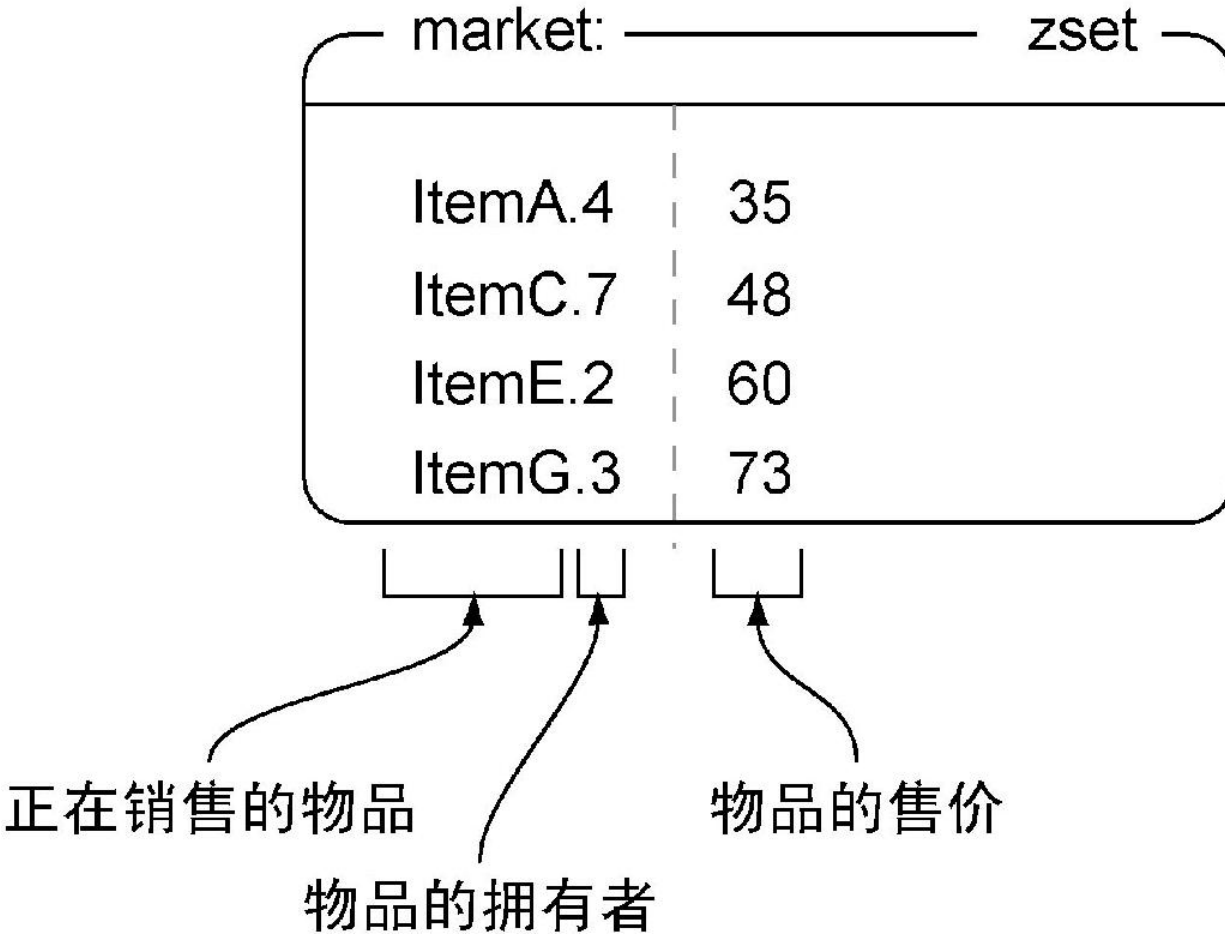


图 4-3 在 Redis 中存储正在销售的商品及其价格

redis 4.0 版本开始，redis 支持了 WATCH 命令，用于实现分布式锁。

4.4.2 分布式锁

redis 4.0 版本开始，redis 支持了 WATCH 命令，用于实现分布式锁。WATCH 命令用于监视一个或多个 key，一旦 key 被修改，WATCH 命令就会返回失败。EXEC 命令用于执行一个事务，UNWATCH 命令用于取消监视。WATCH 命令可以用于实现分布式锁，具体实现如下：

1. 使用 WATCH 命令监视 key。

2. 使用 EXEC 命令执行事务。

3. 如果 key 被修改，EXEC 命令会返回失败。

4. 使用 UNWATCH 命令取消监视。

5. 使用 DISCARD 命令取消事务。

6. 使用 WATCH 命令监视 key。

7. 使用 EXEC 命令执行事务。

8. 如果 key 被修改，EXEC 命令会返回失败。

9. 使用 UNWATCH 命令取消监视。

10. 使用 DISCARD 命令取消事务。

redis 4.0 版本开始，redis 支持了 WATCH 命令，用于实现分布式锁。WATCH 命令用于监视一个或多个 key，一旦 key 被修改，WATCH 命令就会返回失败。EXEC 命令用于执行一个事务，UNWATCH 命令用于取消监视。WATCH 命令可以用于实现分布式锁，具体实现如下：

redis 4.0 版本开始，redis 支持了 WATCH 命令，用于实现分布式锁。

```

def list_item(conn, itemid, sellerid, price):
    inventory = "inventory:%s"%sellerid
    item = "%s.%s"%(itemid, sellerid)
    end = time.time() + 5
    pipe = conn.pipeline()
    while time.time() < end:
        try:
            pipe.watch(inventory)
            if not pipe.sismember(inventory, itemid):
                pipe.unwatch()
                return None

            pipe.multi()
            pipe.zadd("market:", item, price)
            pipe.srem(inventory, itemid)
            pipe.execute()
            return True
        except redis.exceptions.WatchError:
            pass
    return False

```

如果指定的商品不在用户的包裹里面，那么停止对包裹键的监视并返回一个空值。

把被销售的商品添加到商品买卖市场里面。

监视用户包裹发生的变化。

检查用户是否仍然持有将要被销售的商品。

如果执行 execute 方法没有引发 WatchError 异常，那么说明事务执行成功，并且对包裹键的监视也已经结束。

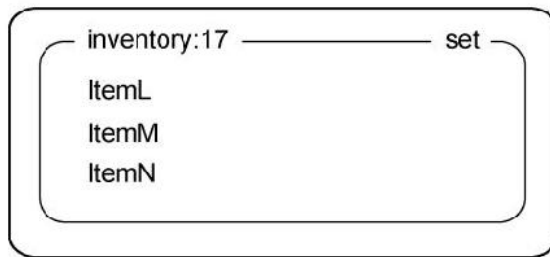
用户的包裹已经发生了变化，重试。

list_item() 函数实现了一个 while 循环，循环中调用 WATCH 命令监视包裹键。如果指定的商品不在用户的包裹里面，那么停止对包裹键的监视并返回一个空值。如果指定的商品在用户的包裹里面，那么调用 MULTI 命令开始事务，调用 ZADD 命令将商品添加到商品买卖市场，调用 SREM 命令将商品从包裹中移除，最后调用 EXECUTE 命令执行事务。如果事务执行成功，那么返回 True，否则返回 False。

4-4 Frank ID 17 97 ItemM

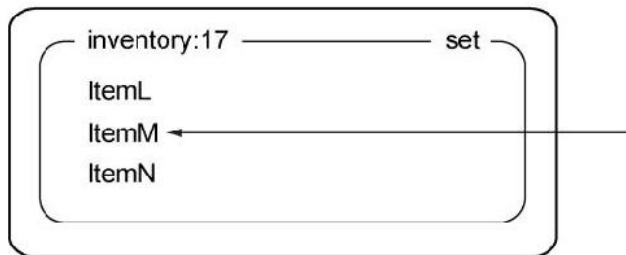
list_item()

```
watch('inventory:17')
```

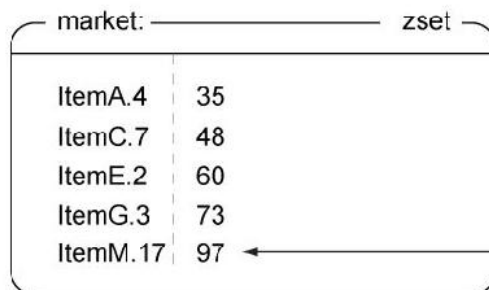


监视包裹发生的任何变化。

```
sismember('inventory:17', 'ItemM')
```

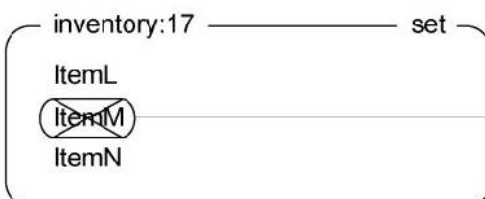


确保被销售的物品仍然存在于 Frank 的包裹里面。



```
zadd('market:', 'ItemM.17', 97)
```

```
srem('inventory:17', 'ItemM')
```



因为没有 Redis 命令可以在移除集合元素的同时，将被移除的元素改名并添加到有序集合里面，所以这里使用了 `ZADD` 和 `SREM` 两个命令来实现这一操作。

```
list_item(conn, "ItemM", 17, 97)
```

この関数は、指定したオブジェクトの値が変更されたときに、
WATCHオブジェクトのEXECメソッドを呼び出して、
指定した処理を実行します。

この関数は、次のように使います。

4.4.3 例

図4-6のpurchase_item()関数は、WATCHオブジェクトの
WATCHメソッドを使って、purchase_item()関数の
実行時に、指定した処理を実行します。
この関数は、WatchError例外を発生させ、
10秒間待ちます。

図4-6 purchase_item()関数

```

def purchase_item(conn, buyerid, itemid, sellerid, lprice):
    buyer = "users:%s"%buyerid
    seller = "users:%s"%sellerid
    item = "%s.%s"%(itemid, sellerid)
    inventory = "inventory:%s"%buyerid
    end = time.time() + 10
    pipe = conn.pipeline()

    while time.time() < end:
        try:
            pipe.watch("market:", buyer)

            price = pipe.zscore("market:", item)
            funds = int(pipe.hget(buyer, "funds"))
            if price != lprice or price > funds:
                pipe.unwatch()
                return None

            pipe.multi()
            pipe.hincrby(seller, "funds", int(price))
            pipe.hincrby(buyer, "funds", int(-price))
            pipe.sadd(inventory, itemid)
            pipe.zrem("market:", item)
            pipe.execute()
            return True
        except redis.exceptions.WatchError:
            pass

    return False

```

对商品交易市场以及买家的个人信息进行监视。

检查买家想要购买的商品的价格是否出现了变化，以及买家是否有足够的钱来购买这件商品。

先将买家支付的钱转移给卖家，然后将被购买的商品移交给买家。

如果买家的个人信息或者商品交易市场在交易的过程中出现了变化，那么进行重试。

```

#####

#####

#####

#####

#####

```

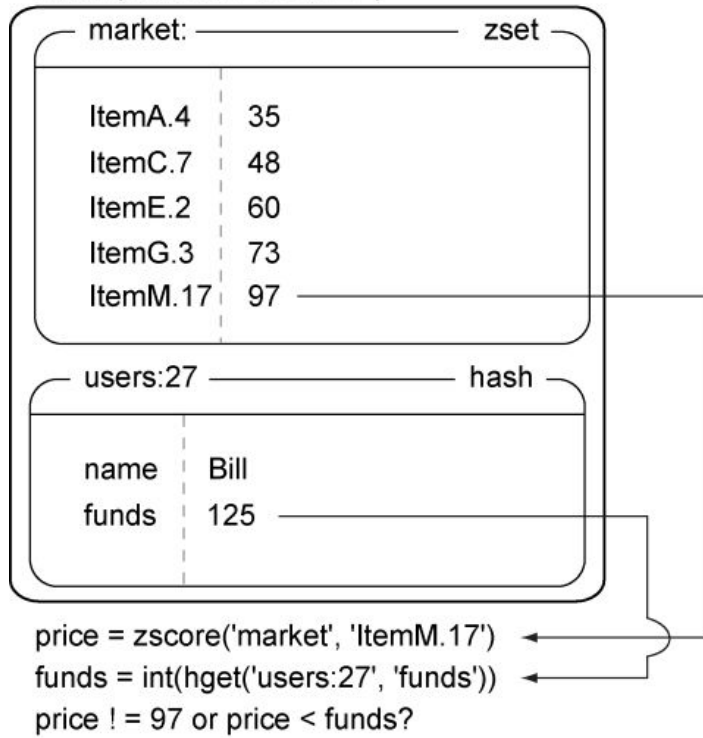
```

#####Bill#####ID#27#####Frank#####

ItemM#4-5#4-6#####

```

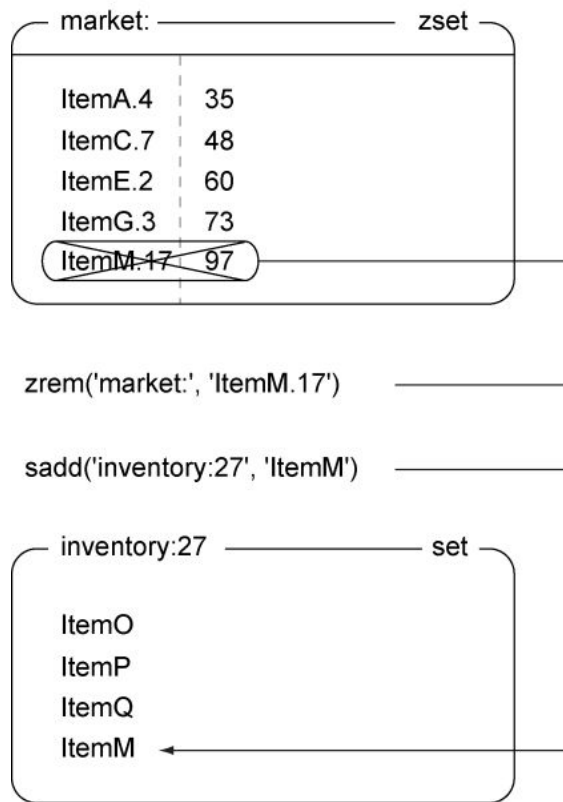
```
watch('market:', 'users:27')
```



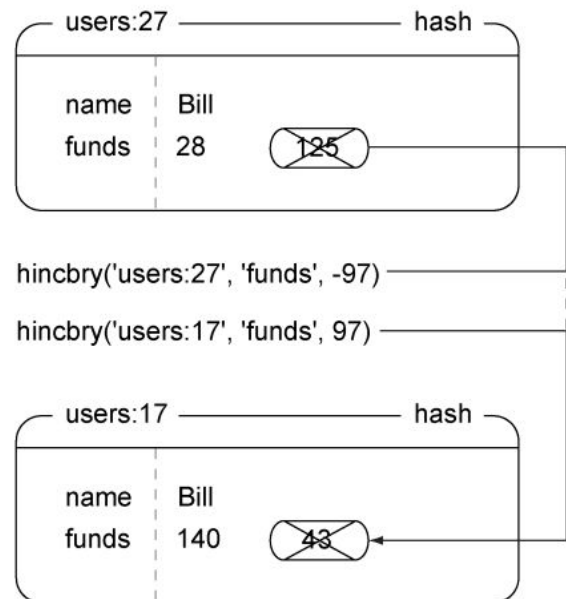
对物品买卖市场以及Bill
的个人信息进行监视。

验证物品的售价是否并未改变，
以及Bill是否有足够的钱来购买
该物品。

4-5



将物品移动到Bill的包裹里面。



将Bill支付的货款转移给Frank。

图4-6 使用Redis实现购物车和支付功能

图4-6展示了使用Redis实现购物车和支付功能的示意图。左侧部分展示了购物车的实现，使用ZSET数据结构存储商品及其价格，使用SET数据结构存储购物车中的商品。右侧部分展示了支付功能的实现，使用HASH数据结构存储用户信息，包括姓名和余额。通过Redis的原子操作，实现了将商品从购物车移动到购物车，并将用户的支付金额转移到目标用户账户。

Redis是一个高性能的键值对数据库，支持多种数据类型，如字符串、哈希、列表、集合等。在实现购物车和支付功能时，Redis提供了高效的原子操作，确保了数据的一致性和完整性。相比之下，SQL数据库在处理并发操作时，需要使用SELECT FOR UPDATE来锁定数据，这可能会导致性能瓶颈。

COMMIT ROLLBACK
Redis
WATCH Redis
WATCH optimistic locking
pessimistic locking
——

Redis
WATCH Redis
WATCH optimistic locking
pessimistic locking
——

WATCH MULTI EXEC
7

ePUBw.COM ePUBw.COM

4.5

03MULTIEXEC“”——MULTIEXEC

```

2#####MGET#####MSET#####HMGET#####
HMSET#####RPUSH#####LPUSH#####SADD#####ZADD#####
#####non-
transactional pipeline#####

```

[illegible]

```
pipe = conn.pipeline()
```

```

    pipeline() == True
MULTIEXEC pipeline()

```

False 返回 False，MULTI 返回 EXEC，Redis 返回 pipeline() 返回 False，Redis 返回 pipeline() 返回 False。

2.1~2.5 更新 update_token() 函数，将 cookie 4-7 更新为 2.5 版本，update_token() 函数在 2~5 版本 Redis 中，Redis 2~5 版本。

4-7 2.5 版本 update_token() 函数

创建令牌与已登录用户之间的映射。	<pre>def update_token(conn, token, user, item=None): timestamp = time.time() conn.hset('login:', token, user) conn.zadd('recent:', token, timestamp) if item: conn.zadd('viewed:' + token, item, timestamp) conn.zremrangebyrank('viewed:' + token, 0, -26) conn.zincrby('viewed:', item, -1)</pre>	获取时间戳。
把用户浏览过的商品记录起来。		记录令牌最后一次出现的时间。
移除旧商品，只记录最新浏览的 25 件商品。		更新给定商品的被浏览次数。

Redis Web 2~5 版本 update_token() 函数在 2~10 版本 Web 100~500 版本 update_token() 函数在 4-8 版本 update_token_pipeline() 函数

图4-8 update_token_pipeline()

```
def update_token_pipeline(conn, token, user, item=None):
    timestamp = time.time()
    pipe = conn.pipeline(False)
    pipe.hset('login:', token, user)
    pipe.zadd('recent:', token, timestamp)
    if item:
        pipe.zadd('viewed:' + token, item, timestamp)
        pipe.zremrangebyrank('viewed:' + token, 0, -26)
        pipe.zincrby('viewed:', item, -1)
    pipe.execute()
```

← 设置流水线。

← 执行那些被流水线包裹的命令。

图4-8 Redis 流水线性能测试

update_token_pipeline() 函数在 Redis 中执行 1000 次

Web 应用调用 update_token_pipeline() 函数

Web 应用调用 500 到 1000 次

update_token_pipeline() 函数

图4-9 update_token() 函数

update_token_pipeline() 函数在 Redis 中执行 1000 次

Web 应用调用 Redis 函数 4-9 次

Web 应用调用 update_token() 函数

update_token_pipeline() 函数

图4-9 benchmark_update_token()

```
def benchmark_update_token(conn, duration):
    for function in (update_token, update_token_pipeline):
        count = 0
        start = time.time()
        end = start + duration
        while time.time() < end:
            count += 1
            function(conn, 'token', 'user', 'item')
        delta = time.time() - start
        print function.__name__, count, delta, count / delta
```

测试会分别执行 update_token() 函数和 update_token_pipeline() 函数。

计算函数的执行时长。

打印测试结果。

设置计数器以及测试结束的条件。

调用两个函数的其中一个。

图 4-4 性能测试脚本

表 4-4 性能测试脚本测试结果/Redis 性能测试

测试环境	测试数据	测试时间	update_table() 性能	update_table_pipeline() 性能
测试环境 Unix 系统	1Gb gigabit	0.015ms	3 761	6 394
测试环境 Linux	1Gb	0.015ms	3 257	5 991
测试环境 Windows	1Gb	0.271ms	739	2 841
测试环境 VPN 测试	1.8Mb megabit	48ms	3.67	18.2

4-4 5
4 Python
Redis 4.6

Redis
Redis standard

ePUBw.COM ePUBw.COM

4.6

```

RedisRedis
Redis
Redis

```

```

RedisRedisRedisRedisRedisRedisRedisRedisRedisRedis
Redisredis-benchmark4-10redis-benchmarkRedis

```

redis-benchmark

```
$ redis-benchmark -c 1 -q
PING (inline): 34246.57 requests per second
PING: 34843.21 requests per second
MSET (10 keys): 24213.08 requests per second
SET: 32467.53 requests per second
GET: 33112.59 requests per second
INCR: 32679.74 requests per second
LPUSH: 33333.33 requests per second
LPOP: 33670.04 requests per second
SADD: 33222.59 requests per second
SPOP: 34482.76 requests per second
LPUSH (again, in order to bench LRANGE): 33222.59 requests per second
LRANGE (first 100 elements): 22988.51 requests per second
LRANGE (first 300 elements): 13888.89 requests per second
LRANGE (first 450 elements): 11061.95 requests per second
LRANGE (first 600 elements): 9041.59 requests per second
```

redis-benchmark Redis 1
redis-benchmark redis-benchmark
50 redis-benchmark
redis-benchmark

redis-benchmark
redis-benchmark
redis-benchmark
Python redis-benchmark
50% 60%

redis-benchmark 25%
30% "Cannot assign requested address"

4-5 redis-benchmark Python

4-5 Redis redis-benchmark
表 4-5

表头	表头	表头
redis-benchmark 50% 60%	表头	表头

問題箇所	原因	対策
redis-benchmarkで25%~30%	redis-benchmarkの実行環境がRedisの稼働環境と異なる	Redisの稼働環境でredis-benchmarkを実行する
“Cannot assign requested address”エラーが発生する	Redisの稼働環境でredis-benchmarkを実行する	Redisの稼働環境でredis-benchmarkを実行する

4-5 Redisの稼働環境でredis-benchmarkを実行する

Redisの稼働環境でredis-benchmarkを実行する

4-5 Redisの稼働環境でredis-benchmarkを実行する

1.4 Redisの稼働環境でredis-benchmarkを実行する

Redisの稼働環境でredis-benchmarkを実行する

Python Redisの稼働環境でredis-benchmarkを実行する

redis.Redis() Redisの稼働環境でredis-benchmarkを実行する

Redisの稼働環境でredis-benchmarkを実行する

Python Redisの稼働環境でredis-benchmarkを実行する

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

4.7 总结

Redis 是一个开源的、高性能的、分布式的键值数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持持久化、主从复制、集群部署等功能。Redis 的应用场景非常广泛，如缓存、消息队列、分布式锁等。

Redis 的持久化方式主要有两种：AOF（Append Only File）和 RDB（Redis Database Backup）。AOF 通过记录 Redis 的写操作来持久化数据，而 RDB 则是通过快照的方式将内存中的数据写入磁盘。Redis 还支持 WATCH、MULTI、EXEC 等命令来实现事务操作。

Redis 的 WATCH、MULTI、EXEC 命令可以用于实现分布式锁。Redis 的 Redis 命令可以用于实现分布式锁。Redis 的 Redis 命令可以用于实现分布式锁。Redis 的 Redis 命令可以用于实现分布式锁。

① Redis 是一个开源的、高性能的、分布式的键值数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持持久化、主从复制、集群部署等功能。Redis 的应用场景非常广泛，如缓存、消息队列、分布式锁等。

② ACID 是指 atomicity（原子性）、consistency（一致性）、isolation（隔离性）、durability（持久性）。Redis 支持 ACID 事务。

③ Redis 支持 B 树索引。Redis 支持 B 树索引。Redis 支持 B 树索引。Redis 支持 B 树索引。

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

第5章 Redis数据库

本章主要介绍

- Redis数据库
- Redis数据库的安装
- Redis IP地址的设置
- Redis数据库的备份

本章主要介绍Redis数据库的安装、配置、备份和恢复。

Redis数据库是一种基于内存的键值数据库，支持多种数据类型。

本章主要介绍Redis数据库的安装、配置、备份和恢复。

本章主要介绍Redis数据库的安装、配置、备份和恢复。

本章主要介绍Redis数据库的安装、配置、备份和恢复。——component

本章主要介绍Redis数据库的安装、配置、备份和恢复。

本章主要介绍Redis数据库的安装、配置、备份和恢复。

本章主要介绍Redis数据库的安装、配置、备份和恢复。ePUBw.COM

本章主要介绍Redis数据库的安装、配置、备份和恢复。

5.1 Redis 的安装

Redis 是一个开源的、高性能的、分布式的、基于内存的数据库。它支持多种数据类型，如字符串、哈希、列表、集合等。Redis 的安装和配置相对简单，本文将介绍如何在 Linux 和 Unix 系统上安装 Redis。

Linux 和 Unix 系统上安装 Redis 的步骤如下：
1. 安装 Redis 的依赖库。Redis 依赖于 libjemalloc、liblua 和 libz 等库。可以使用以下命令安装这些库：
yum install jemalloc lua libz
2. 编译并安装 Redis。从 Redis 官方网站下载源代码，并运行以下命令进行编译和安装：
tar xzf redis-*.tar.gz
cd redis-*/
make
make install
3. 配置 Redis。Redis 的配置文件位于 /etc/redis/redis.conf。可以根据需要进行配置，如设置监听端口、密码等。
4. 启动 Redis。使用以下命令启动 Redis 服务：
redis-server /etc/redis/redis.conf
5. 验证 Redis 是否安装成功。使用以下命令启动 Redis 客户端，并输入 ping 命令：
redis-cli
ping
如果返回 PONG，则表示 Redis 安装成功。

syslog 是一个系统日志守护进程，用于收集和分发系统日志。它支持多种日志格式，如文本、JSON 等。Linux 和 Unix 系统上安装 syslog 的步骤如下：
1. 安装 syslog 的依赖库。syslog 依赖于 libz、libcrypto 和 libssl 等库。可以使用以下命令安装这些库：
yum install zlib openssl
2. 编译并安装 syslog。从 syslog 官方网站下载源代码，并运行以下命令进行编译和安装：
tar xzf syslog-*.tar.gz
cd syslog-*/
./configure
make
make install
3. 配置 syslog。syslog 的配置文件位于 /etc/syslog.conf。可以根据需要进行配置，如设置日志格式、日志文件等。
4. 启动 syslog。使用以下命令启动 syslog 服务：
systemctl start syslogd
5. 验证 syslog 是否安装成功。使用以下命令启动 syslog 客户端，并输入 test 命令：
syslogd
test
如果返回 OK，则表示 syslog 安装成功。

syslog 是一个系统日志守护进程，用于收集和分发系统日志。它支持多种日志格式，如文本、JSON 等。Linux 和 Unix 系统上安装 syslog 的步骤如下：
1. 安装 syslog 的依赖库。syslog 依赖于 libz、libcrypto 和 libssl 等库。可以使用以下命令安装这些库：
yum install zlib openssl
2. 编译并安装 syslog。从 syslog 官方网站下载源代码，并运行以下命令进行编译和安装：
tar xzf syslog-*.tar.gz
cd syslog-*/
./configure
make
make install
3. 配置 syslog。syslog 的配置文件位于 /etc/syslog.conf。可以根据需要进行配置，如设置日志格式、日志文件等。
4. 启动 syslog。使用以下命令启动 syslog 服务：
systemctl start syslogd
5. 验证 syslog 是否安装成功。使用以下命令启动 syslog 客户端，并输入 test 命令：
syslogd
test
如果返回 OK，则表示 syslog 安装成功。

Redis 数据库的日志记录功能，可以通过 Redis 的 `syslog` 模块来实现。该模块允许 Redis 将日志消息发送到系统日志（syslog）中。这通常用于在 Redis 出现问题时记录错误信息，以便进行故障排查。

`syslog` 模块在 Redis 的配置文件 `redis.conf` 中启用。通过配置 `syslog` 模块，Redis 可以将日志消息发送到指定的 syslog 服务。此外，Redis 还支持 `time-sensitive log`（时间敏感日志），这允许 Redis 根据日志消息的时间戳来记录最近的消息（recent log message）。

5.1.1 配置

要启用 Redis 的 `syslog` 模块，需要在 `redis.conf` 文件中配置以下参数：

- `syslog`：指定 Redis 是否启用 syslog 模块。
- `syslogname`：指定 Redis 在 syslog 中的名称。
- `syslogfacility`：指定 Redis 在 syslog 中的设施（facility）。

图 5-1 展示了 `log_recent()` 函数在 Redis 数据库中的工作原理。该函数通过 `LUSH`（Log User Session Hash）和 `LRange`（Log Range）命令来记录最近的消息。具体来说，`LUSH` 命令用于将最近的消息添加到 Redis 的 `log_recent` 列表中，而 `LRange` 命令则用于从该列表中获取最近的消息。

图 5-1 `log_recent()` 函数

```

SEVERITY = {
    logging.DEBUG: 'debug',
    logging.INFO: 'info',
    logging.WARNING: 'warning',
    logging.ERROR: 'error',
    logging.CRITICAL: 'critical',
}
SEVERITY.update({name, name} for name in SEVERITY.values())

def log_recent(conn, name, message, severity=logging.INFO, pipe=None):
    severity = str(SEVERITY.get(severity, severity)).lower()
    destination = 'recent:%s:%s'%(name, severity)
    message = time.asctime() + ' ' + message
    pipe = pipe or conn.pipeline()
    pipe.lpush(destination, message)
    pipe.ltrim(destination, 0, 99)
    pipe.execute()

```

将当前时间添加到消息里面，用于记录消息的发送时间。
 设置一个字典，将大部分日志的安全级别映射为字符串。
 尝试将日志的安全级别转换为简单的字符串。
 使用流水线来将通信往返次数降低为一次。
 对日志列表进行修剪，让它只包含最新的100条消息。
 创建负责存储消息的键。
 将消息添加到日志列表的最前面。
 执行两个命令。

```

#####info#####debug#####
log_recent()#####——#####LPUSH#####LTRIM#####
#####

```

5.1.2 管道

```

#####log_recent()#####log_recent()
#####
#####
#####
#####

```

```

#####5-2#####log_common()#####
#####
#####
#####
#####

```

5-2 log_common()

```

def log_common(conn, name, message, severity=logging.INFO, timeout=5):
    severity = str(SEVERITY.get(severity, severity)).lower()
    destination = 'common:%s:%s'%(name, severity)
    start_key = destination + ':start'
    pipe = conn.pipeline()
    end = time.time() + timeout
    while time.time() < end:
        try:
            pipe.watch(start_key)
            now = datetime.utcnow().timetuple()
            hour_start = datetime(*now[:4]).isoformat()
            existing = pipe.get(start_key)
            pipe.multi()
            if existing and existing < hour_start:
                pipe.rename(destination, destination + ':last')
                pipe.rename(start_key, destination + ':pstart')
                pipe.set(start_key, hour_start)
            elif not existing:
                pipe.set(start_key, hour_start)
            pipe.zincrby(destination, message)
            log_recent(pipe, name, message, severity, pipe)
            return
        except redis.exceptions.WatchError:
            continue
    log_recent(pipe, name, message, severity, pipe)

```

设置日志的安全级别。

对记录当前小时数的键进行监视，确保轮换操作可以正确地执行。

……那么将这些旧的常见日志消息归档。

log_recent() 函数负责记录日志并调用 execute() 函数。

负责存储近期的常见日志消息的键。

因为程序每小时需要轮换一次日志，所以它使用一个键来记录当前所处的小时数。

取得当前时间。

取得当前所处的小时数。

创建一个事务。

如果这个常见日志消息列表记录的是上一个小时的日志……

更新当前所处的小时数。

对记录日志出现次数的计数器执行自增操作。

如果程序因为其他客户端正在执行归档操作而出现监视错误，那么进行重试。

```

    redis_log_recent(log_level);
    if (redis_log_level == WATCH/MULTI/EXEC) {
        redis_log_recent(log_level);
    }
    Redis *redis = redis_get_redis();
    return redis;
}
Redis *redis_get_redis()
{
    Redis *redis = NULL;
    redis = redis_new();
    return redis;
}

```


□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

5.2

[illegible][illegible]

```
Redis Redis  
time series counter
```

5.2.1 Redis

计数器使用 Redis 的 incr 命令来增加计数。在 Redis 中，计数器是一个字符串，它存储了当前的计数值。当需要增加计数时，使用 incr 命令，Redis 会自动将字符串解析为整数并增加 1。例如，如果当前计数是 120，执行 incr 后，计数会变成 121。

在 Redis 中，计数器是一个字符串，它存储了当前的计数值。当需要增加计数时，使用 incr 命令，Redis 会自动将字符串解析为整数并增加 1。

1. 计数器

计数器使用 Redis 的 incr 命令来增加计数。在 Redis 中，计数器是一个字符串，它存储了当前的计数值。当需要增加计数时，使用 incr 命令，Redis 会自动将字符串解析为整数并增加 1。例如，如果当前计数是 120，执行 incr 后，计数会变成 121。

count:5:hits		hash
1336376410	45	
1336376405	28	
1336376395	17	
1336376400	29	
...	...	

这个计数器显示网站在 2012 年 5 月 7 日早晨 7:39:55 到 7:40:00 总共获得了 17 次点击。

图 5-1 计数器显示 2012 年 5 月 7 日 7:40 计数器显示 5 次点击

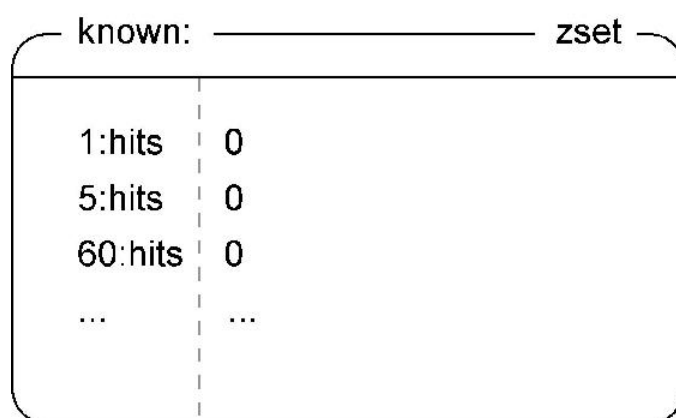
有序集合（ordered sequence）是 Redis 中一种特殊的数据类型。它的特点是集合中的成员都是唯一的，并且成员会根据其分值（score）进行排序。当两个成员的分值相同时，Redis 会根据成员名（lexicographically）进行排序。

在 Redis 中，有序集合的创建和更新操作如下：

1. 创建有序集合：使用 `zadd` 命令。

2. 更新有序集合：使用 `zadd` 命令，如果成员已经存在，则会更新其分值。

图 5-2 展示了有序集合的存储结构。



当有序集合中的分值都相等时，Redis 将根据成员名来进行排序。

图 5-2 有序集合的存储结构

在 Redis 中，有序集合的创建和更新操作如下：

1. 创建有序集合：使用 `zadd` 命令。

2. 更新有序集合：使用 `zadd` 命令，如果成员已经存在，则会更新其分值。

图 5-3 展示了有序集合的更新操作。

图 5-3 update_counter()

通过取得当前时间来判断应该对哪个时间片执行自增操作。

为我们记录的每种精度都创建一个计数器。

将计数器的引用信息添加到有序集合里面,并将其分值设置为0,以便在之后执行清理操作。

以秒为单位的计数器精度,分别为1秒、5秒、1分钟、5分钟、1小时、5小时、1天——用户可以按需调整这些精度。

```
PRECISION = [1, 5, 60, 300, 3600, 18000, 86400]
```

```
def update_counter(conn, name, count=1, now=None):  
    now = now or time.time()  
    pipe = conn.pipeline()  
    for prec in PRECISION:  
        pnow = int(now / prec) * prec  
        hash = '%s:%s'%(prec, name)  
        pipe.zadd('known:', hash, 0)  
        pipe.hincrby('count:' + hash, pnow, count)  
    pipe.execute()
```

对给定名字和精度的计数器进行更新。

为了保证之后的清理工作可以正确地执行,这里需要创建一个事务型流水线。

取得当前时间片的开始时间。

创建负责存储计数信息的散列。

```
#####ZADD#####HINCRBY#####  
#####5-4#####  
#####HGETALL#####  
#####
```

#####5-4 get_counter()#####

取得存储计数器数据的键的名字。

```
def get_counter(conn, name, precision):  
    hash = '%s:%s'%(precision, name)  
    data = conn.hgetall('count:' + hash)  
    to_return = []  
    for key, value in data.iteritems():  
        to_return.append((int(key), int(value)))  
    to_return.sort()  
    return to_return
```

从 Redis 里面取出计数器数据。

将计数器数据转换成指定的格式。

对数据进行排序,把旧的数据样本排在前面。

```
get_counter()#####  
#####  
#####
```

2#####

Redis 的过期策略是惰性（lazy）的，即不会定期去扫描并删除过期的键，而是在访问键的时候才去判断是否过期。如果键没有过期，则返回键的值；如果键已经过期，则删除该键并返回 nil。

Redis 的 EXPIRE 命令用于设置键的过期时间。该命令的语法如下：

```
EXPIRE key seconds
```

process clean up 是指 Redis 在后台运行的清理工作，包括删除过期的键。

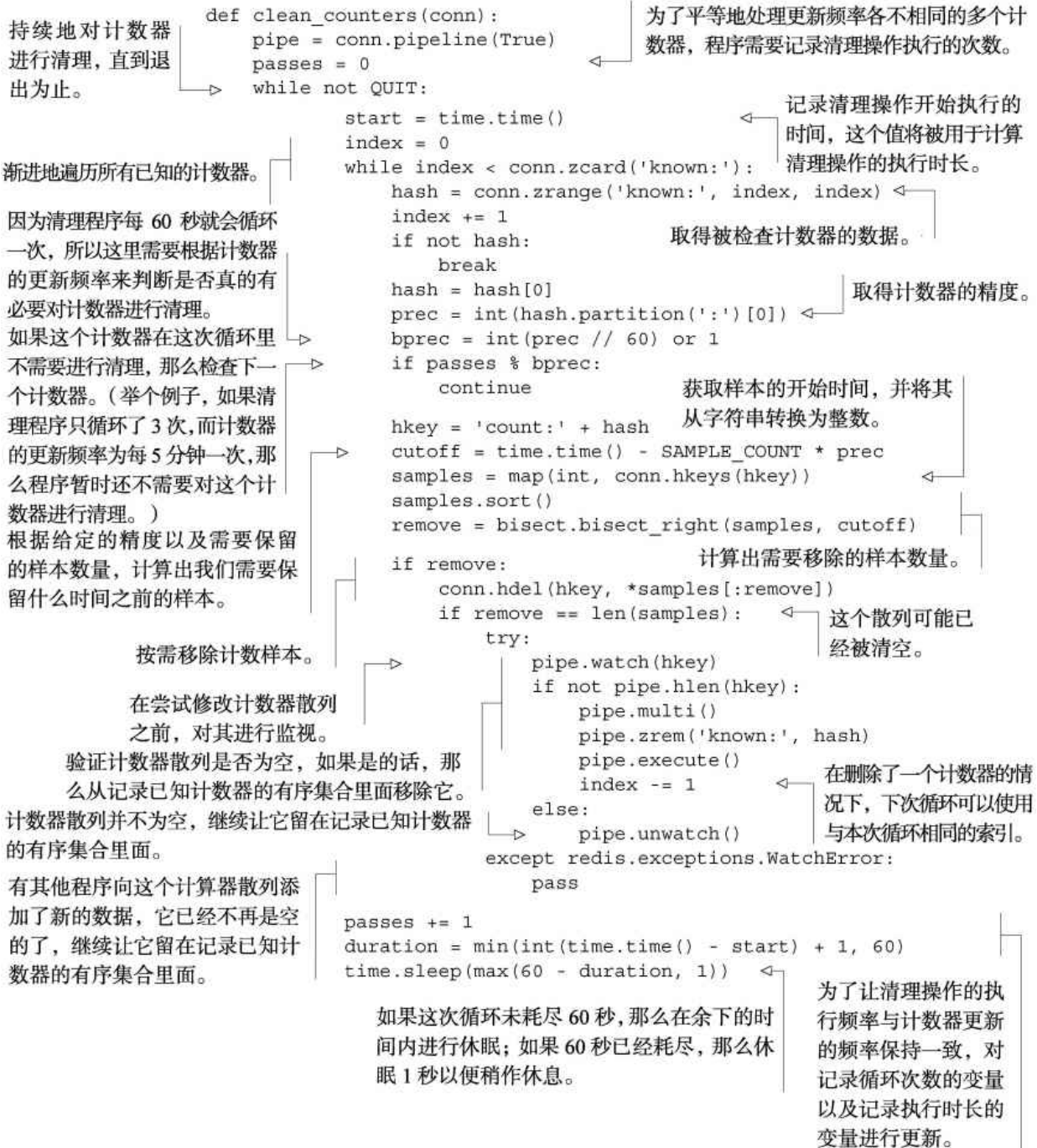
- Redis 的过期策略是惰性的，即不会定期去扫描并删除过期的键，而是在访问键的时候才去判断是否过期。
- Redis 的 EXPIRE 命令用于设置键的过期时间。
- Redis 的过期时间是以秒为单位的。
- Redis 的过期时间可以设置为 0，表示永不过期。

Redis 的过期策略是惰性的，即不会定期去扫描并删除过期的键，而是在访问键的时候才去判断是否过期。如果键没有过期，则返回键的值；如果键已经过期，则删除该键并返回 nil。

5
 5
 5

ZRANGE
 120
 5-5

5-5 clean_counters()



clean_counters()

redis 5.0 版本开始支持对 zset 类型的统计操作

redis 5.0 版本开始支持对 context 类型的统计操作，包括 min、max、count、sum、sumsq、min、max、count、sum、sumsq、MIN、MAX、5-3、ProfilePage、AccessTime 等。

stats:ProfilePage:AccessTime		zset
min	0.035	
max	4.958	
sumsq	194.268	
sum	258.973	
count	2323	

5-3 redis 5.0 版本开始支持对 zset 类型的统计操作

redis 5.0 版本开始支持对 context 类型的统计操作，包括 min、max、count、sum、sumsq、min、max、count、sum、sumsq、MIN、MAX、5-6、ProfilePage、AccessTime 等。

```

redis> SET stats '{"count":1,"sum":1,"sumsq":1}'
redis> ZUNIONSTORE stats 1 MIN MAX
redis> WATCH stats
redis> ZINCRBY stats count
redis> sumsq 3

```

5-6 update_stats()

```

def update_stats(conn, context, type, value, timeout=5):
    destination = 'stats:%s:%s'%(context, type)
    start_key = destination + ':start'
    pipe = conn.pipeline(True)
    end = time.time() + timeout
    while time.time() < end:
        try:
            pipe.watch(start_key)
            now = datetime.utcnow().timetuple()
            hour_start = datetime(*now[:4]).isoformat()

            existing = pipe.get(start_key)
            pipe.multi()
            if existing and existing < hour_start:
                pipe.rename(destination, destination + ':last')
                pipe.rename(start_key, destination + ':pstart')
                pipe.set(start_key, hour_start)

            tkey1 = str(uuid.uuid4())
            tkey2 = str(uuid.uuid4())
            pipe.zadd(tkey1, 'min', value)
            pipe.zadd(tkey2, 'max', value)
            pipe.zunionstore(destination,
                [destination, tkey1], aggregate='min')
            pipe.zunionstore(destination,
                [destination, tkey2], aggregate='max')

            pipe.delete(tkey1, tkey2)
            pipe.zincrby(destination, 'count')
            pipe.zincrby(destination, 'sum', value)
            pipe.zincrby(destination, 'sumsq', value*value)

            return pipe.execute()[-3:]
        except redis.exceptions.WatchError:
            continue

    if new hour started and old data is archived,
    then retry.

```

负责存储统计数据的键。

像common_log()函数一样，处理当前这一个小时的数据和上一个小时的数据。

将值添加到临时键里面。

使用聚合函数 MIN 和 MAX，对存储统计数据的键以及两个临时键进行并集计算。

删除临时键。

返回基本的计数信息，以便函数调用者在有需要时做进一步的处理。

对有序集中的样本数量、值的和、值的平方之和3个成员进行更新。

update_status() 5.1.2

log_common() update_status()

ZUNIONSTORE

update_status()

```

5-7 计算平均值和标准差的函数
def get_stats(conn, context, type):
    key = 'stats:%s:%s'%(context, type)
    data = dict(conn.zrange(key, 0, -1, withscores=True))
    data['average'] = data['sum'] / data['count']
    numerator = data['sumsq'] - data['sum'] ** 2 / data['count']
    data['stddev'] = (numerator / (data['count'] - 1 or 1)) ** .5
    return data

```

5-7 get_status()

```

def get_stats(conn, context, type):
    key = 'stats:%s:%s'%(context, type)
    data = dict(conn.zrange(key, 0, -1, withscores=True))
    data['average'] = data['sum'] / data['count']
    numerator = data['sumsq'] - data['sum'] ** 2 / data['count']
    data['stddev'] = (numerator / (data['count'] - 1 or 1)) ** .5
    return data

```

程序将从这个键里面取出统计数据。

获取基本的统计数据,并将它们都放到一个字典里面。

完成标准差的计算工作。

计算平均值。

计算标准差的第一个步骤。

```

get_stats()

```

5.2.3 Redis

```

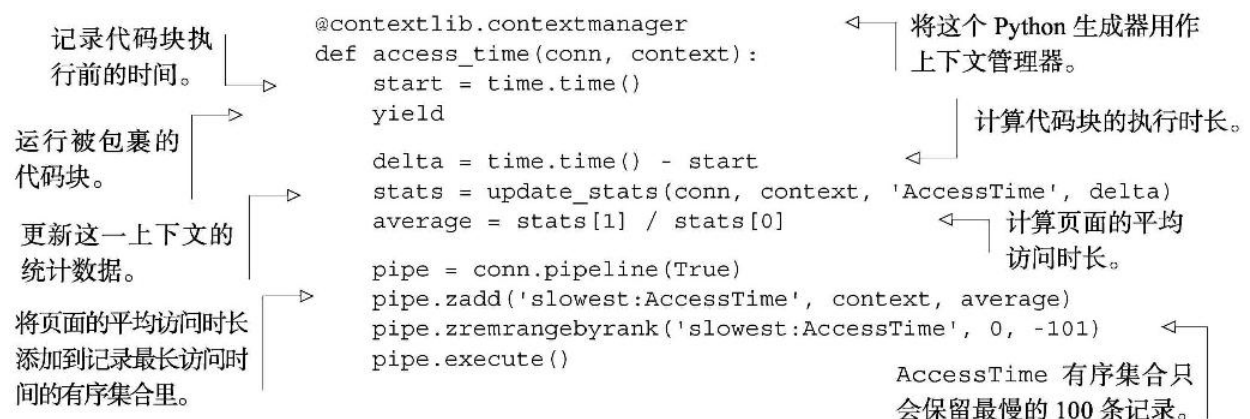
Redis

```

在 Redis 中，我们使用 Redis 的 pipeline 功能来批量执行命令，以提高效率。在 Python 中，我们使用 contextlib 模块的 contextmanager 装饰器来定义上下文管理器。在 Redis 中，我们使用 Redis 的 pipeline 功能来批量执行命令，以提高效率。

在 Python 中，context manager 是一个用于管理上下文的对象。在 Redis 中，我们使用 Redis 的 pipeline 功能来批量执行命令，以提高效率。在 Python 中，context manager 是一个用于管理上下文的对象。

5-8 access_time()



在 Python 中，context manager 是一个用于管理上下文的对象。在 Redis 中，我们使用 Redis 的 pipeline 功能来批量执行命令，以提高效率。在 Python 中，context manager 是一个用于管理上下文的对象。

计算并记录访问时长的上下文管理器就是这样包围代码块的。

```
def process_view(conn, callback):  
    with access_time(conn, request.path):  
        return callback()
```

这个视图 (view) 接受一个 Redis 连接以及一个生成内容的回调函数为参数。

当上下文管理器中的 yield 语句被执行时，这个语句就会被执行。

#####recent_log()#####

#####Graphite#####
#####

#####Redis#####
#####

#####ePUBw.COM#####ePUBw.COM#####

#####

5.3 IP地址地理定位

我们使用Redis来存储IP地址的地理位置信息。Redis是一个键值对数据库，支持多种数据类型。我们使用Redis的GeoStore模块来实现IP地址的地理位置定位。GeoStore模块允许我们存储地理位置信息，并支持对地理位置信息的查询和计算。我们使用Redis的GeoStore模块来存储IP地址的地理位置信息，并支持对地理位置信息的查询和计算。

我们使用Fake Game来模拟地理位置信息。Fake Game是一个模拟地理位置信息的工具，它允许我们生成随机的地理位置信息。我们使用Fake Game来生成随机的地理位置信息，并将其存储在Redis中。我们使用Fake Game来生成随机的地理位置信息，并将其存储在Redis中。

我们使用Redis来存储IP地址的地理位置信息。Redis是一个键值对数据库，支持多种数据类型。我们使用Redis的GeoStore模块来实现IP地址的地理位置定位。GeoStore模块允许我们存储地理位置信息，并支持对地理位置信息的查询和计算。我们使用Redis的GeoStore模块来存储IP地址的地理位置信息，并支持对地理位置信息的查询和计算。

5.3.1 数据源

我们使用IP地址的地理位置信息。我们使用<http://dev.maxmind.com/geoip/geolite>来获取IP地址的地理位置信息。我们使用<http://dev.maxmind.com/geoip/geolite>来获取IP地址的地理位置信息。我们使用<http://dev.maxmind.com/geoip/geolite>来获取IP地址的地理位置信息。


```
def import_ips_to_redis(conn, filename):
    csv_file = csv.reader(open(filename, 'rb'))
    for count, row in enumerate(csv_file):
        start_ip = row[0] if row else ''
        if 'i' in start_ip.lower():
            continue
        if '.' in start_ip:
            start_ip = ip_to_score(start_ip)
        elif start_ip.isdigit():
            start_ip = int(start_ip, 10)
        else:
            continue

        city_id = row[2] + '_' + str(count)
        conn.zadd('ip2cityid:', city_id, start_ip)

    将城市 ID 及其对应的 IP 地址
    分值添加到有序集合里面。
```

这个函数在执行时需要输入
GeoLiteCity-Blocks.csv 文件
所在的路径。

按需将 IP 地址
转换为分值。

略过文件的第一行以
及格式不正确的条目。

构建唯一城市 ID。

import_ips_to_redis() 将 IP 地址添加到 Redis 有序集合
5-11 将城市 ID 添加到 Redis 有序集合
JSON 格式数据

5-11 import_cities_to_redis()

```
def import_cities_to_redis(conn, filename):
    for row in csv.reader(open(filename, 'rb')):
        if len(row) < 4 or not row[0].isdigit():
            continue
        row = [i.decode('latin-1') for i in row]
        city_id = row[0]
        country = row[1]
        region = row[2]
        city = row[3]
        conn.hset('cityid2city:', city_id,
                json.dumps([city, region, country]))
```

这个函数在执行时需要输入
GeoLiteCity-Location.csv 文件
所在的路径。

准备好需要添加到散列
里面的信息。

将城市信息添加
到 Redis 里面。

Redis 有序集合 IP 地址

5.3.2 IP 地址

通过 IP 地址来查找城市 ID 和 IP 地址

beginning 参数表示从 IP 地址的哪个位置开始查找

ip_to_score() 函数将 IP 地址转换为分值以便执行 ZREVRANGEBYSCORE 命令。

ZREVRANGEBYSCORE 命令的 START 和 NUM 参数分别表示从哪个位置开始查找以及查找多少个 ID。

5-12 查找唯一城市 ID

IP 地址转换为分值以便执行 ZREVRANGEBYSCORE 命令。

5-12 find_city_by_ip()

```
def find_city_by_ip(conn, ip_address):  
    if isinstance(ip_address, str):  
        ip_address = ip_to_score(ip_address)  
  
    city_id = conn.zrevrangebyscore(  
        'ip2cityid:', ip_address, 0, start=0, num=1)  
  
    if not city_id:  
        return None  
  
    city_id = city_id[0].partition('_')[0]  
    return json.loads(conn.hget('cityid2city:', city_id))
```

将 IP 地址转换为分值以便执行 ZREVRANGEBYSCORE 命令。

查找唯一城市 ID。

将唯一城市 ID 转换为普通城市 ID。

从散列里面取出城市信息。

find_city_by_ip() 函数将 IP 地址转换为分值以便执行 ZREVRANGEBYSCORE 命令。

“ip2cityid:” 是 Redis 数据库中的散列名称。

7 表示从散列中取出 7 个元素。

Redis 数据库中的散列名称为 service。

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

5.4 Redis 配置

Redis 配置文件的默认名称是 `redis.conf`，位于 Redis 安装目录的 `etc` 子目录下。在启动 Redis 服务时，可以通过 `--config` 选项指定配置文件的路径。如果未指定，则使用默认配置文件。

Redis 配置文件中包含了许多配置项，用于控制 Redis 服务器的行为。这些配置项可以分为几类：网络配置、持久化配置、性能优化配置等。在配置文件中，配置项通常以 `#` 开头，表示注释。要启用某个配置项，只需将 `#` 去掉即可。

Redis 支持 `live configuration`，即可以在运行过程中动态修改配置。通过 `CONFIG SET` 命令，可以设置或修改配置项。例如，可以动态地设置 Redis 的端口号。

5.4.1 Redis 配置

Redis 配置文件中包含了许多配置项，用于控制 Redis 服务器的行为。这些配置项可以分为几类：网络配置、持久化配置、性能优化配置等。在配置文件中，配置项通常以 `#` 开头，表示注释。要启用某个配置项，只需将 `#` 去掉即可。

例如，配置 Redis 的持久化选项。在配置文件中，可以找到 `save` 配置项，用于设置 Redis 的持久化策略。可以通过设置 `save` 配置项来控制 Redis 何时将数据写入磁盘。

Web
Redis

Redis
Redis

2
is_under_maintenance()
True
False
is-under-maintenance
is-under-maintenance
True
False
Redis
Web
is_under_maintenance()
5-13
is_under_maintenance()

5-13 is_under_maintenance()

```
LAST_CHECKED = None
IS_UNDER_MAINTENANCE = False

def is_under_maintenance(conn):
    global LAST_CHECKED, IS_UNDER_MAINTENANCE

    if LAST_CHECKED < time.time() - 1:
        LAST_CHECKED = time.time()
        IS_UNDER_MAINTENANCE = bool(
            conn.get('is-under-maintenance'))

    return IS_UNDER_MAINTENANCE
```

距离上次检查是否已经超过 1 秒?

检查系统是否正在进行维护。

将两个变量设置为全局变量以便在之后对它们进行写入。

更新最后检查时间。

返回一个布尔值，用于表示系统是否正在进行维护。

is_under_maintenance() plug into Redis Web is_under_maintenance() commonly accessible location Redis

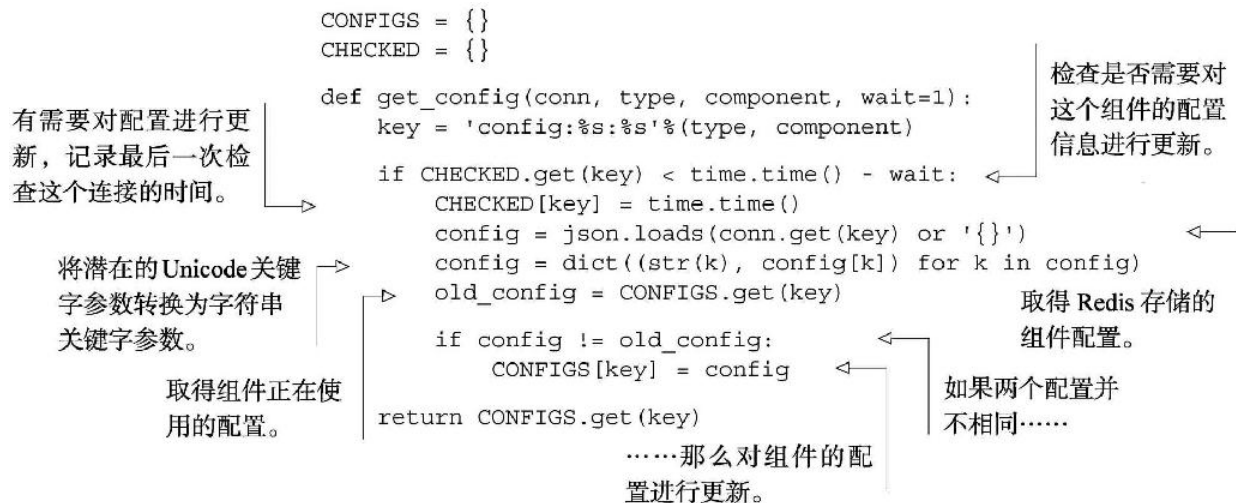
5.4.2 Redis

Redis Redis Redis

Redis cookies Redis Redis “ ” key space Redis Redis Redis

def get_config() 每隔0.1秒检查一次配置信息

5-15 get_config()



Redis 配置信息存储在 Redis 数据库中，通过 Redis 客户端进行访问。

5.4.3 Redis 配置

Redis 配置信息存储在 Redis 数据库中，通过 Redis 客户端进行访问。

Redis 配置信息存储在 Redis 数据库中，通过 Redis 客户端进行访问。

Redis 配置信息存储在 Redis 数据库中，通过 Redis 客户端进行访问。

Redis 配置信息存储在 Redis 数据库中，通过 Redis 客户端进行访问。

Python 的装饰器（Decorator）是 Python 语言中一种重要的编程技巧，它允许我们在不修改被装饰函数本身的情况下，动态地添加功能。本文将介绍 Python 装饰器的基本概念、使用方法和应用场景。

在 5-16 节中，我们将通过一个具体的例子来展示装饰器的使用。这个例子涉及 Redis 连接的管理，我们将使用装饰器来封装 Redis 连接相关的逻辑，使得代码更加简洁和易于维护。

5-16 redis_connection() 函数

因为函数每次被调用都需要获取这个配置键，所以我们干脆把它缓存起来。

包装器接受一个函数作为参数，并使用另一个函数来包裹这个函数。

如果有旧配置存在，那么获取它。

如果有新配置存在，那么获取它。

对配置进行处理并将其用于创建 Redis 连接。

将 Redis 连接以及其他匹配的参数传递给被包裹函数，然后调用该函数并返回它的执行结果。

```
REDIS_CONNECTIONS = {}

def redis_connection(component, wait=1):
    key = 'config:redis:' + component
    def wrapper(function):
        @functools.wraps(function)
        def call(*args, **kwargs):
            old_config = CONFIGS.get(key, object())
            _config = get_config(
                config_connection, 'redis', component, wait)

            config = {}
            for k, v in _config.iteritems():
                config[k.encode('utf-8')] = v

            if config != old_config:
                REDIS_CONNECTIONS[key] = redis.Redis(**config)

            return function(
                REDIS_CONNECTIONS.get(key), *args, **kwargs)
        return call
    return wrapper
```

将应用组件的名字传递给装饰器。

将被包裹函数的一些有用的元数据复制给配置处理器。

创建负责管理连接信息的函数。

如果新旧配置并不相同，那么创建新的连接。

返回被包裹的函数。

返回用于包裹 Redis 函数的包装器。

函数*args**kwargs 是Python中非常常用的两个参数，
args是positional argument，kwargs是named argument。
关于Python的更多细节，请访问<http://mng.bz/KM5x>。

图5-16展示了如何使用redis_connection()函数来装饰log_recent()函数。
调用redis_connection()函数时，它会将log_recent()函数的调用者
作为参数传递给Redis函数。因此，调用redis_connection()函数时，
图5-17展示了redis_connection()函数在5.1.1版本中的
log_recent()函数。

图5-17 使用log_recent()函数

这个函数的定义和之前展示的一样，没有发生任何变化。

```
@redis_connection('logs')
def log_recent(conn, app, message):
    'the old log_recent() code'

log_recent('main', 'User 235 logged in')
```

redis_connection()装饰器非常容易使用。

我们再也不必在调用log_recent()函数时手动地向它传递日志服务器的连接了。

图5-17展示了如何使用“log_recent()”函数来装饰log_recent()函数。
“log_recent()”函数在5.1.1版本中。
请访问<http://www.python.org/dev/peps/pep-0318/>。

redis_connection()log_recent()
redis_connection()5.2.3
access_time()Redis
redis_connection()

ePUBw.COM ePUBw.COM

5.5 配置

Redis 的配置文件 redis.conf 位于安装目录的 etc 子目录下。默认情况下，Redis 使用 6379 端口。在配置文件中，可以设置 Redis 的端口、密码、持久化策略等。在配置文件中，还可以设置 Redis 的日志文件、主从复制等。在配置文件中，还可以设置 Redis 的内存限制、超时时间等。

Redis 的配置文件 redis.conf 位于安装目录的 etc 子目录下。默认情况下，Redis 使用 6379 端口。在配置文件中，可以设置 Redis 的端口、密码、持久化策略等。在配置文件中，还可以设置 Redis 的日志文件、主从复制等。在配置文件中，还可以设置 Redis 的内存限制、超时时间等。

① Python 的 Redis 客户端 redis-py 位于安装目录的 lib 子目录下。在配置文件中，可以设置 Redis 的端口、密码、持久化策略等。在配置文件中，还可以设置 Redis 的日志文件、主从复制等。在配置文件中，还可以设置 Redis 的内存限制、超时时间等。

本站所有资源均来自网络，如有侵权，请联系删除。
ePUBw.COM 本站所有资源均来自网络，如有侵权，请联系删除。

本站所有资源均来自网络，如有侵权，请联系删除。

第6章 Redis部署方案

本章主要介绍

- Redis部署方案
- Redis部署方案
- Redis部署方案
- Redis部署方案
- Redis部署方案
- Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

本章主要介绍

本章主要介绍Redis部署方案

本章主要介绍Redis部署方案

6.1 問題

Webブラウザのautocomplete機能は、ユーザーの入力に基づいて、検索履歴や一般的な検索語を提案する。例えば、Googleで「Betty White」と入力すると、検索履歴に「Betty White」がある場合、autocompleteが「Betty White」を提案する。これは、Webブラウザのautocomplete機能が、Googleの検索履歴を記憶しているためである。Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。

Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。

6.1.1 問題の背景

Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。Googleは、ユーザーの検索履歴を記憶し、それをautocompleteに利用している。これは、Googleのプライバシーポリシーに違反している。

100
 6-1

聊天对象：

je
最近联系过的人…… Jean Jeannie Jeff

6-1 /e

100
 Redis
 Redis
 Redis
 Redis
 Python
 Python

Redis 3
 3

1. 检查联系人是否存在

2. 如果存在，则删除他

3. 将联系人推入列表的最前端，并保留列表中的前100个联系人

3. 检查联系人是否存在，如果存在，则删除他。如果不存在，则将联系人推入列表的最前端。并保留列表中的前100个联系人。

6-1 add_update_contact()

```
def add_update_contact(conn, user, contact):
    ac_list = 'recent:' + user
    pipeline = conn.pipeline(True)
    pipeline.lrem(ac_list, contact)
    pipeline.lpush(ac_list, contact)
    pipeline.ltrim(ac_list, 0, 99)
    pipeline.execute()
```

如果联系人已经存在，那么移除他。

只保留列表里面的前100个联系人。

准备执行原子操作。

将联系人推入列表的最前端。

实际地执行以上操作。

6-1 add_update_contact()

6-1 add_update_contact()

```
def remove_contact(conn, user, contact):
    conn.lrem('recent:' + user, contact)
```

Python 6-2

6-2 fetch_autocomplete_list()

```
def fetch_autocomplete_list(conn, user, prefix):
    candidates = conn.lrange('recent:' + user, 0, -1)
    matches = []
    for candidate in candidates:
        if candidate.lower().startswith(prefix):
            matches.append(candidate)
    return matches
```

检查每个候选
联系人。

发现一个匹配的联系人。

获取自动补全
列表。

返回所有匹配的联系人。

fetch_autocomplete_list()

“”

100

most-
recently-used list
least-recently-used list

6.1.2

Redis Redis
Redis Redis
Redis Redis

Fake Game
Fake Game
Fake Game
Fake Game

Redis Redis
Redis Redis
Redis Redis

Redis Redis
Redis Redis
Redis Redis
Redis Redis
Redis Redis
Redis Redis


```

valid_characters = '`abcdefghijklmnopqrstuvwxyz{'
def find_prefix_range(prefix):
    posn = bisect.bisect_left(valid_characters, prefix[-1:])
    suffix = valid_characters[(posn or 1) - 1]
    return prefix[:-1] + suffix + '{', prefix + '{'

```

找到前驱字符。

准备一个由已知字符组成的列表。

在字符列表中查找前缀字符所处的位置。

返回范围。

在字符列表中查找前缀字符所处的位置。
 `find_prefix_range()`
 在字符列表中查找前缀字符所处的位置。

在字符列表中查找前缀字符所处的位置。
 `a`
`z`
 在字符列表中查找前缀字符所处的位置。

在字符列表中查找前缀字符所处的位置。
 `UTF-8`
`UTF-16`
`UTF-32`
 在字符列表中查找前缀字符所处的位置。

在字符列表中查找前缀字符所处的位置。
 `UTF-8`
`UTF-16`
`UTF-32`
`null`
 在字符列表中查找前缀字符所处的位置。

在字符列表中查找前缀字符所处的位置。
 `10`
 在字符列表中查找前缀字符所处的位置。

128个UUID
WATCHMULTIEXEC
6-4

6-4 autocomplete_on_prefix()

```
def autocomplete_on_prefix(conn, guild, prefix):
    start, end = find_prefix_range(prefix)
    identifier = str(uuid.uuid4())
    start += identifier
    end += identifier
    zset_name = 'members:' + guild
    conn.zadd(zset_name, start, 0, end, 0)
    pipeline = conn.pipeline(True)
    while 1:
        try:
            pipeline.watch(zset_name)
            sindex = pipeline.zrank(zset_name, start)
            eindex = pipeline.zrank(zset_name, end)
            erange = min(sindex + 9, eindex - 2)
            pipeline.multi()
            pipeline.zrem(zset_name, start, end)
            pipeline.zrange(zset_name, sindex, erange)
            items = pipeline.execute()[-1]
            break
        except redis.exceptions.WatchError:
            continue
    return [item for item in items if '{' not in item]
```

将范围的起始元素和结束元素添加到有序集合里面。

找到两个被插入元素在有序集合中的排名。

获取范围内的值，然后删除之前插入的起始元素和结束元素。

如果有其他自动补全操作正在执行，那么从获取到的元素里面移除起始元素和结束元素。

根据给定的前缀计算出查找范围的起点和终点。

如果自动补全有序集合已经被其他客户端修改过了，那么重试。

autocomplete_on_prefix()

WATCH

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

6.2 乐观锁

乐观锁“假设”数据不会被其他线程修改，因此不需要在操作前 acquire 锁，操作完成后 release 锁。Redis 使用 shared-memory data structure 实现乐观锁，通过 WATCH 命令监视数据，如果数据在操作前被其他线程修改，则操作失败。Redis 使用 WATCH 命令实现乐观锁，通过 SETNX 命令实现乐观锁。

乐观锁“假设”数据不会被其他线程修改，因此不需要在操作前 acquire 锁，操作完成后 release 锁。Redis 使用 WATCH 命令监视数据，如果数据在操作前被其他线程修改，则操作失败。Redis 使用 WATCH 命令实现乐观锁，通过 SETNX 命令实现乐观锁。

乐观锁“假设”数据不会被其他线程修改，因此不需要在操作前 acquire 锁，操作完成后 release 锁。Redis 使用 WATCH 命令监视数据，如果数据在操作前被其他线程修改，则操作失败。Redis 使用 WATCH 命令实现乐观锁，通过 SETNX 命令实现乐观锁。

redis 的 WATCH 命令是乐观锁，它通过监视一个或多个键，如果这些键在事务执行前被其他客户端修改，那么事务就会失败。WATCH 命令的语法如下：

6.2.1 乐观锁

redis 的 WATCH 命令是乐观锁，它通过监视一个或多个键，如果这些键在事务执行前被其他客户端修改，那么事务就会失败。WATCH 命令的语法如下：

EXEC 命令会执行 4.6 版本引入的 WATCH 命令，MULTI 命令会执行 EXEC 命令。WATCH 命令的语法如下：

ID 命令 ID 命令的语法如下：

6-2 乐观锁的示意图

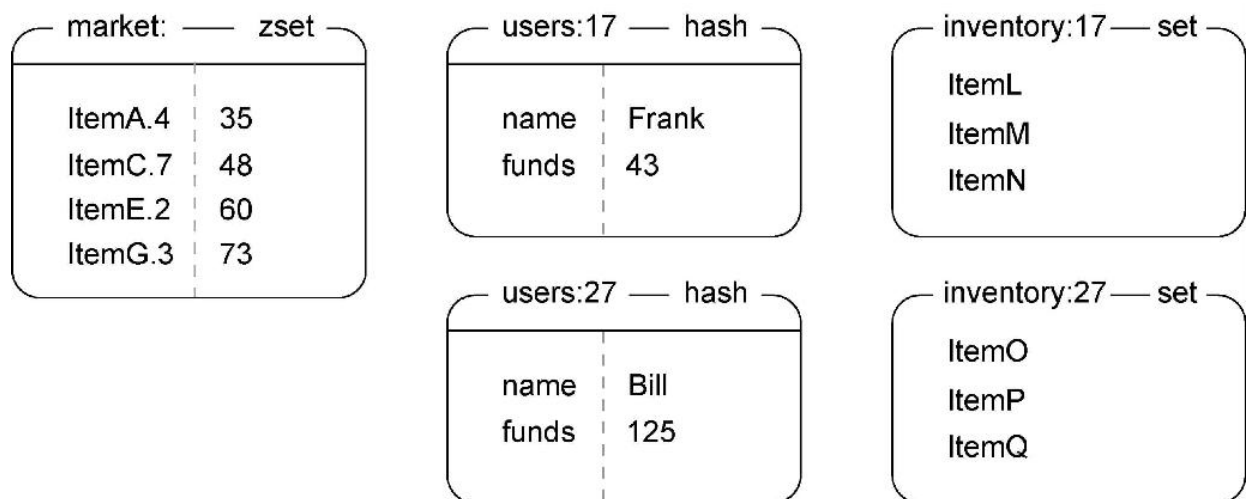


图 6-2 乐观锁的示意图

该图展示了 Redis 数据库中的四个数据集合，每个集合都包含一些数据。这些集合分别是：

- market: zset (有序集合)
- users:17 hash (哈希表)
- inventory:17 set (集合)
- users:27 hash (哈希表)
- inventory:27 set (集合)

每个集合都包含一些数据，例如 market 集合包含 ItemA.4 (35)、ItemC.7 (48)、ItemE.2 (60) 和 ItemG.3 (73)。users 集合包含 name 和 funds 字段。inventory 集合包含 ItemL、ItemM、ItemN、ItemO、ItemP 和 ItemQ。

redis 的 WATCH 命令是乐观锁，它通过监视一个或多个键，如果这些键在事务执行前被其他客户端修改，那么事务就会失败。WATCH 命令的语法如下：


```
def purchase_item(conn, buyerid, itemid, sellerid, lprice):
    #...
    pipe.watch("market:", buyer)

    price = pipe.zscore("market:", item)
    funds = int(pipe.hget(buyer, 'funds'))
    if price != lprice or price > funds:
        pipe.unwatch()
        return None
    pipe.multi()
    pipe.hincrby(seller, 'funds', int(price))
    pipe.hincrby(buyerid, 'funds', int(-price))
    pipe.sadd(inventory, itemid)
    pipe.zrem("market:", item)
    pipe.execute()
    return True

#...
```

监视市场以及买家个人信息发生的变化。

检查商品是否已经售出、商品的价格是否已经发生了变化，以及买家是否有足够的钱来购买这件商品。

将买家支付的钱转移给卖家，并将卖出的商品转移给买家。

6-6 list_item() 6-7
purchase_item() WATCH MULTI EXEC

3 3
1 1 5 1 5
5 6-1

6-1 60

1 1	145 000	27 000	80 000	14 ms
5 1	331 000	<200	50 000	150 ms

	数据库大小	事务大小	事务数	事务执行时间
5000005000	206 000	<600	161 000	498 ms

图6-1展示了事务执行的性能测试结果。在3000000次事务执行中，平均事务大小为206 000字节，事务执行时间小于600微秒，事务执行速率为161 000次/秒，事务执行总时间为498 ms。事务执行时间包括WATCH、MULTI、EXEC三个命令的执行时间。WATCH命令用于监视数据，MULTI命令用于开始事务，EXEC命令用于执行事务。WATCH命令的执行时间非常短，MULTI命令的执行时间也非常短，EXEC命令的执行时间也非常短。WATCH命令的执行时间非常短，MULTI命令的执行时间也非常短，EXEC命令的执行时间也非常短。

6.2.2 锁

Redis提供了两种锁：分布式锁和分布式锁。分布式锁是一种用于分布式系统中的锁，它可以在多个节点上同时使用。分布式锁的实现原理是利用Redis的SETNX命令来实现。SETNX命令用于设置键值对，如果键不存在，则设置成功。SETNX命令的执行时间非常短，因此可以用于实现分布式锁。SETNX命令的执行时间非常短，因此可以用于实现分布式锁。

Redis提供了两种锁：分布式锁和分布式锁。分布式锁是一种用于分布式系统中的锁，它可以在多个节点上同时使用。分布式锁的实现原理是利用Redis的SETNX命令来实现。SETNX命令用于设置键值对，如果键不存在，则设置成功。SETNX命令的执行时间非常短，因此可以用于实现分布式锁。SETNX命令的执行时间非常短，因此可以用于实现分布式锁。

Redis提供了两种锁：分布式锁和分布式锁。分布式锁是一种用于分布式系统中的锁，它可以在多个节点上同时使用。分布式锁的实现原理是利用Redis的SETNX命令来实现。SETNX命令用于设置键值对，如果键不存在，则设置成功。SETNX命令的执行时间非常短，因此可以用于实现分布式锁。SETNX命令的执行时间非常短，因此可以用于实现分布式锁。

Redis 的持久化策略

- Redis 的持久化策略分为两种：快照持久化和 AOF 持久化。
 - 快照持久化：Redis 会定期将内存中的数据快照保存到磁盘上。快照的大小取决于内存中数据的大小。快照的生成时间取决于数据的大小和 Redis 的配置。
 - AOF 持久化：Redis 会将所有的写操作都记录到 AOF 文件中。AOF 文件的大小取决于写操作的数量。AOF 文件的生成时间取决于写操作的数量和 Redis 的配置。
- Redis 的持久化策略还可以配置为混合持久化。混合持久化结合了快照持久化和 AOF 持久化的优点。快照持久化可以保证数据的完整性，而 AOF 持久化可以保证数据的持久性。
- Redis 的持久化策略还可以配置为只读模式。只读模式下的 Redis 不会持久化数据，只会在内存中保存数据。
- Redis 的持久化策略还可以配置为只写模式。只写模式下的 Redis 只会持久化写操作，而不会持久化读操作。

Redis 的持久化策略可以配置为快照持久化，快照的大小可以配置为 100 000，快照的生成时间可以配置为 225 000。Redis 的持久化策略还可以配置为 AOF 持久化，AOF 文件的大小可以配置为 100 000，AOF 文件的生成时间可以配置为 225 000。Redis 的持久化策略还可以配置为混合持久化，混合持久化结合了快照持久化和 AOF 持久化的优点。

6.2.3 Redis 的持久化策略

Redis 的持久化策略分为两种：快照持久化和 AOF 持久化。快照持久化是指 Redis 会定期将内存中的数据快照保存到磁盘上。AOF 持久化是指 Redis 会将所有的写操作都记录到 AOF 文件中。Redis 的持久化策略还可以配置为混合持久化。混合持久化结合了快照持久化和 AOF 持久化的优点。Redis 的持久化策略还可以配置为只读模式。只读模式下的 Redis 不会持久化数据，只会在内存中保存数据。Redis 的持久化策略还可以配置为只写模式。只写模式下的 Redis 只会持久化写操作，而不会持久化读操作。

Redis 的持久化策略可以配置为快照持久化，快照的大小可以配置为 100 000，快照的生成时间可以配置为 225 000。Redis 的持久化策略还可以配置为 AOF 持久化，AOF 文件的大小可以配置为 100 000，AOF 文件的生成时间可以配置为 225 000。Redis 的持久化策略还可以配置为混合持久化，混合持久化结合了快照持久化和 AOF 持久化的优点。Redis 的持久化策略还可以配置为只读模式。只读模式下的 Redis 不会持久化数据，只会在内存中保存数据。Redis 的持久化策略还可以配置为只写模式。只写模式下的 Redis 只会持久化写操作，而不会持久化读操作。


```
def release_lock(conn, lockname, identifier):
    pipe = conn.pipeline(True)
    lockname = 'lock:' + lockname

    while True:
        try:
            pipe.watch(lockname)
            if pipe.get(lockname) == identifier:
                pipe.multi()
                pipe.delete(lockname)
                pipe.execute()
                return True

            pipe.unwatch()
            break

        except redis.exceptions.WatchError:
            pass

    return False
```

检查进程是否仍然持有锁。

释放锁。

有其他客户端修改了锁，重试。

← 进程已经失去了锁。

在调用 `release_lock()` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `release_lock()` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `release_lock()` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `release_lock()` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `release_lock()` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `WATCH` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `WATCH` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

在调用 `WATCH` 之前，我们首先检查锁是否仍然持有。如果锁仍然持有，我们尝试删除它。如果删除失败，我们退出循环。如果删除成功，我们返回 `True`。如果删除失败，我们退出循环。如果删除失败，我们退出循环。

图6-2 在60秒内释放锁

	锁名称	锁标识	锁过期时间	锁过期时间
--	-----	-----	-------	-------

この表は、監視カメラの動作時間と消費電力を示しています。

監視カメラの動作時間と消費電力

表6-3 監視カメラの動作時間と消費電力

表6-3 監視カメラの動作時間と消費電力

	動作時間	消費電力	動作時間	消費電力
1台あたり1台あたりWATCH	145 000	27 000	80 000	14ms
1台あたり1台あたり	51 000	50 000	0	1ms
1台あたり1台あたり	113 000	110 000	0	<1ms
5台あたり1台あたりWATCH	331 000	<200	50 000	150ms
5台あたり1台あたり	68 000	13 000	<10	5ms
5台あたり1台あたり	192 000	36 000	0	<2ms
5台あたり5台あたりWATCH	206 000	<600	161 000	498ms
5台あたり5台あたり	21 000	20 500	0	14ms

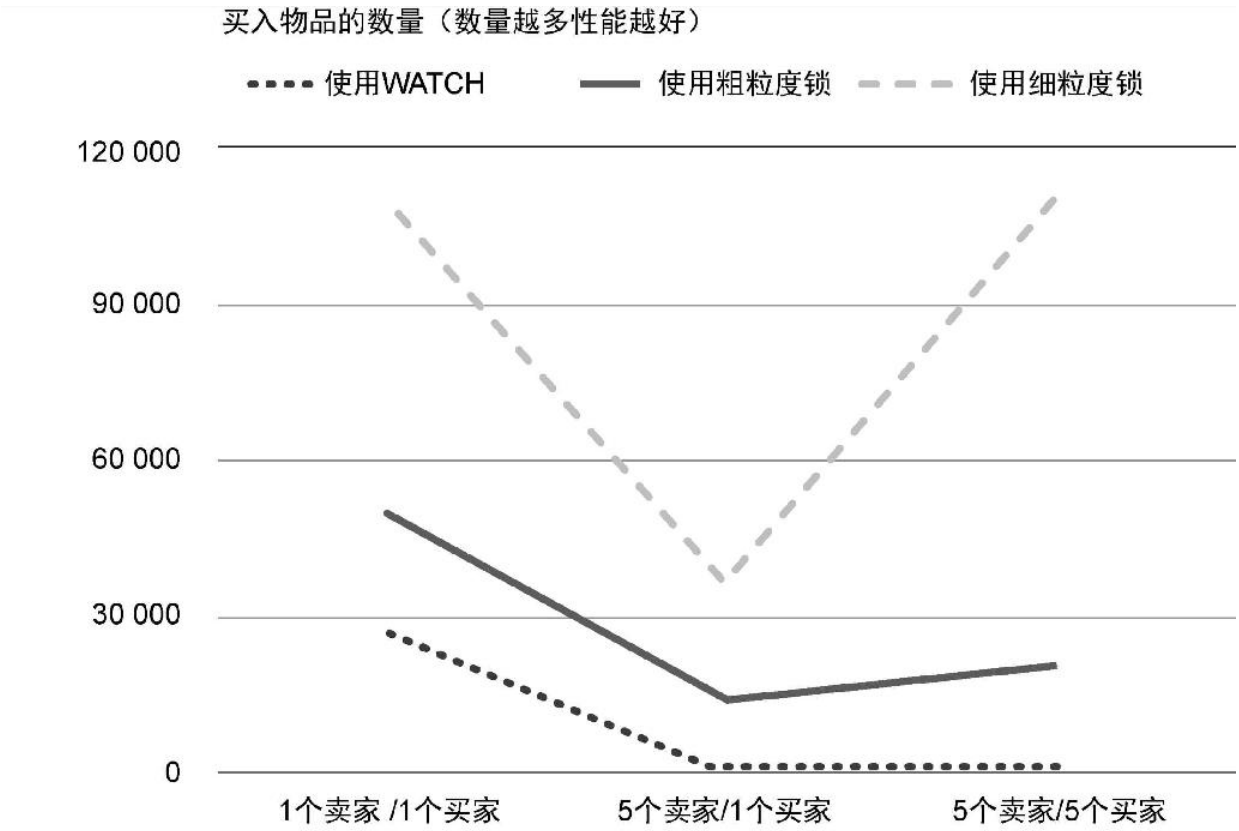


图6-3 在60个物品中，使用WATCH、粗粒度锁、细粒度锁，在5个卖家/1个买家、5个卖家/5个买家的场景下，买入物品的数量（数量越多性能越好）

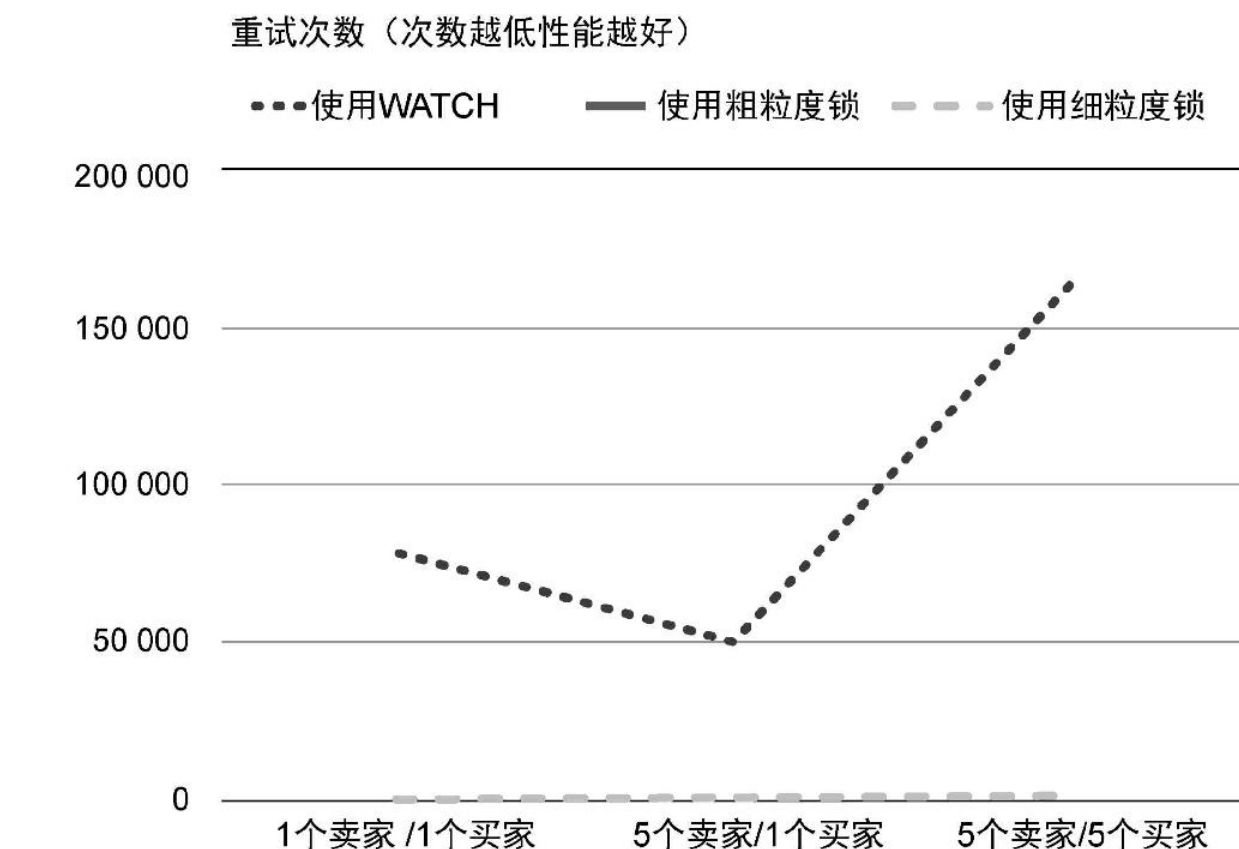


图6-4 60个并发用户同时访问数据库时，使用不同锁机制的重试次数

图6-5 使用WATCH锁机制时，不同并发用户数下的重试次数

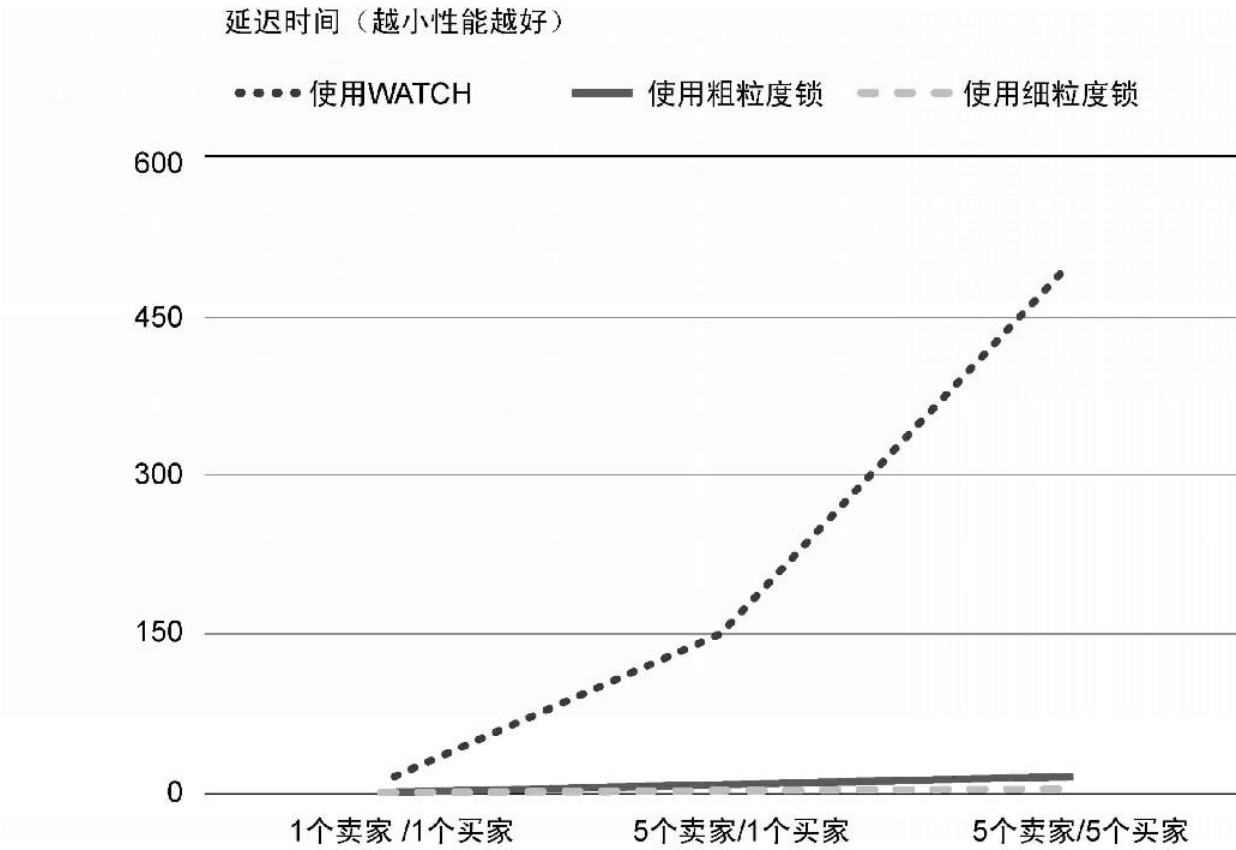


图6-5 在单买家多卖家的场景下，使用WATCH锁的延迟时间比使用粗粒度锁和细粒度锁的延迟时间高得多。WATCH锁的延迟时间约为500 ms，而粗粒度锁和细粒度锁的延迟时间约为14 ms。

在单买家多卖家的场景下，使用WATCH锁的延迟时间比使用粗粒度锁和细粒度锁的延迟时间高得多。WATCH锁的延迟时间约为500 ms，而粗粒度锁和细粒度锁的延迟时间约为14 ms。

在单买家多卖家的场景下，使用WATCH锁的延迟时间比使用粗粒度锁和细粒度锁的延迟时间高得多。WATCH锁的延迟时间约为500 ms，而粗粒度锁和细粒度锁的延迟时间约为14 ms。

Redis 2.8.10 版本开始，Redis 提供了 SET 命令的 NX 选项，用于实现分布式锁。SET 命令的 NX 选项表示只有在 key 不存在的情况下才会设置 value。如果 key 已经存在，则不会设置 value。这可以用于实现分布式锁，因为锁的 key 只能被一个进程设置。如果其他进程尝试设置相同的 key，则会失败。这确保了锁的互斥性。

6.2.5 分布式锁的实现

Redis 提供了 SET 命令的 NX 选项，用于实现分布式锁。SET 命令的 NX 选项表示只有在 key 不存在的情况下才会设置 value。如果 key 已经存在，则不会设置 value。这可以用于实现分布式锁，因为锁的 key 只能被一个进程设置。如果其他进程尝试设置相同的 key，则会失败。这确保了锁的互斥性。

Redis 提供了 SET 命令的 NX 选项，用于实现分布式锁。SET 命令的 NX 选项表示只有在 key 不存在的情况下才会设置 value。如果 key 已经存在，则不会设置 value。这可以用于实现分布式锁，因为锁的 key 只能被一个进程设置。如果其他进程尝试设置相同的 key，则会失败。这确保了锁的互斥性。

Redis 提供了 SET 命令的 NX 选项，用于实现分布式锁。SET 命令的 NX 选项表示只有在 key 不存在的情况下才会设置 value。如果 key 已经存在，则不会设置 value。这可以用于实现分布式锁，因为锁的 key 只能被一个进程设置。如果其他进程尝试设置相同的 key，则会失败。这确保了锁的互斥性。

Redis 6-11 提供了 `acquire_lock()` 和 `acquire_lock_with_timeout()` 两个函数，用于实现分布式锁。

6-11 `acquire_lock_with_timeout()`

```
def acquire_lock_with_timeout(
    conn, lockname, acquire_timeout=10, lock_timeout=10):
    identifier = str(uuid.uuid4())
    lockname = 'lock:' + lockname
    lock_timeout = int(math.ceil(lock_timeout))
    end = time.time() + acquire_timeout
    while time.time() < end:
        if conn.setnx(lockname, identifier):
            conn.expire(lockname, lock_timeout)
            return identifier
        elif not conn.ttl(lockname):
            conn.expire(lockname, lock_timeout)

        time.sleep(.001)
    return False
```

128 位随机标识符。

确保传给 EXPIRE 的都是整数。

获取锁并设置过期时间。

检查过期时间，并在有需要时对其进行更新。

def acquire_lock_with_timeout(conn, lockname, acquire_timeout=10, lock_timeout=10):

identifier = str(uuid.uuid4())

lockname = 'lock:' + lockname

lock_timeout = int(math.ceil(lock_timeout))

end = time.time() + acquire_timeout

while time.time() < end:

if conn.setnx(lockname, identifier):

conn.expire(lockname, lock_timeout)

return identifier

elif not conn.ttl(lockname):

conn.expire(lockname, lock_timeout)

time.sleep(.001)

return False

Redis 2.6.12 中，SET 命令与 SETNX 命令和 SETEX 命令是等价的。

6.1.2 原子性操作

Redis 提供了 WATCH 命令，用于监视一个或多个键，如果在指定的时间范围内，没有任何对监视的键进行修改的操作，那么，就可以对监视的键进行写操作。

WATCH 命令的语法如下：

```
WATCH key1 key2 ... keyN
```

其中，key1、key2、...、keyN 是要监视的键名。

Redis 的 WATCH 命令与 MULTI 命令一起使用，可以保证对监视的键进行写操作时，其他客户端不能同时修改这些键。

EXEC 命令用于执行在 WATCH 命令和 MULTI 命令之间执行的命令。

WATCH 命令和 EXEC 命令一起使用，可以保证对监视的键进行写操作时，其他客户端不能同时修改这些键。

APIRedis
WATCHWATCH

counting
semaphore

ePUBw.COM ePUBw.COM

6.3 线程池

线程池是一种多线程处理形式，会事先创建好固定数量的线程，当有请求到来时，这些线程就会去处理请求。线程池可以避免频繁地创建和销毁线程，从而提高效率。

线程池通常包含以下几个部分：线程池管理器、线程池、线程池工作线程。线程池管理器负责线程池的创建和销毁，线程池负责管理线程池中的线程，线程池工作线程负责处理请求。线程池的大小可以根据需要进行配置，通常设置为5~6个线程。线程池还可以配置超时时间，防止线程长时间占用资源。

线程池的大小可以根据需要进行配置，通常设置为5~6个线程。线程池还可以配置超时时间，防止线程长时间占用资源。

线程池的大小可以根据需要进行配置，通常设置为5~6个线程。线程池还可以配置超时时间，防止线程长时间占用资源。

线程池的大小可以根据需要进行配置，通常设置为5~6个线程。线程池还可以配置超时时间，防止线程长时间占用资源。

6.3.1 线程池的实现

redis 12.1.0 64bit
redis 12.1.0 64bit
redis 12.1.0 64bit
redis 12.1.0 64bit
redis 12.1.0 64bit

Redis
EXPIRE
redis

redis
Unix 6-6
redis

semaphore:remote	zset
c5a58232-0d6c-467f-976f-d577163b24f2	1326437034.113
4cec3d88-a80e-4713-832c-42ec957fb845	1326437037.511
c73f379d-6a5b-48a6-9796-b6f9bbb55402	1326437037.621
40f93169-f8c7-40d3-a342-f0ec68b17dde	1326437038.765

6-6 redis

redis
redis 0

semaphore:remote:owner	zset
8f53c28e-4ec5-4dc3-ad9e-132bd7b1539b	7350
2a0a3d3c-034b-454a-bedd-da79f3284922	7353
183bd37a-6bb9-424b-aaad-34a49454687d	7354
e8bce9c2-f994-46f5-ad86-230e84b1644d	7361

semaphore:remote:counter	string
7361	

图6-7 Redis中的信号量

Redis中的信号量是通过ZSET和STRING两个数据类型实现的。ZSET用于存储信号量的所有者和权重，STRING用于存储信号量的计数器。ZSET的键是semaphore:remote:owner，值是所有者ID和权重。STRING的键是semaphore:remote:counter，值是计数器值。

图6-14展示了acquire_fair_semaphore()函数的实现。该函数首先从Redis中获取信号量的所有者和权重，然后根据权重进行排序，选择权重最小的所有者作为新的所有者。最后，将信号量的计数器加1，并将新的所有者和权重写入Redis。

图6-14 acquire_fair_semaphore()函数


```
def acquire_fair_semaphore(conn, semname, limit, timeout=10):
    identifier = str(uuid.uuid4())
    czset = semname + ':owner'
    ctr = semname + ':counter'

    now = time.time()
    pipeline = conn.pipeline(True)
    pipeline.zremrangebyscore(semname, '-inf', now - timeout)
    pipeline.zinterstore(czset, {czset: 1, semname: 0})

    pipeline.incr(ctr)
    counter = pipeline.execute()[-1]

    pipeline.zadd(semname, identifier, now)
    pipeline.zadd(czset, identifier, counter)

    pipeline.zrank(czset, identifier)
    if pipeline.execute()[-1] < limit:
        return identifier

    pipeline.zrem(semname, identifier)
    pipeline.zrem(czset, identifier)
    pipeline.execute()
    return None
```

128 位随机标识符。

删除超时的信号量。

对计数器执行自增操作，并获取计数器在执行自增操作之后的值。

尝试获取信号量。

通过检查排名来判断客户端是否取得了信号量。

客户端成功取得了信号量。

客户端未能取得信号量，清理无用数据。

`acquire_fair_semaphore()` 的伪代码

```
acquire_semaphore()
    1. 生成一个 128 位的随机 ID
    2. 将 ID 和当前时间戳添加到 Redis 的集合中
    3. 将 ID 和当前计数器值添加到 Redis 的集合中
    4. 返回 ID
```

32 位的随机 ID 和 32 位的 Redis 计数器值

$2^{31}-1$ 的计数器值，32 位的计数器值 2

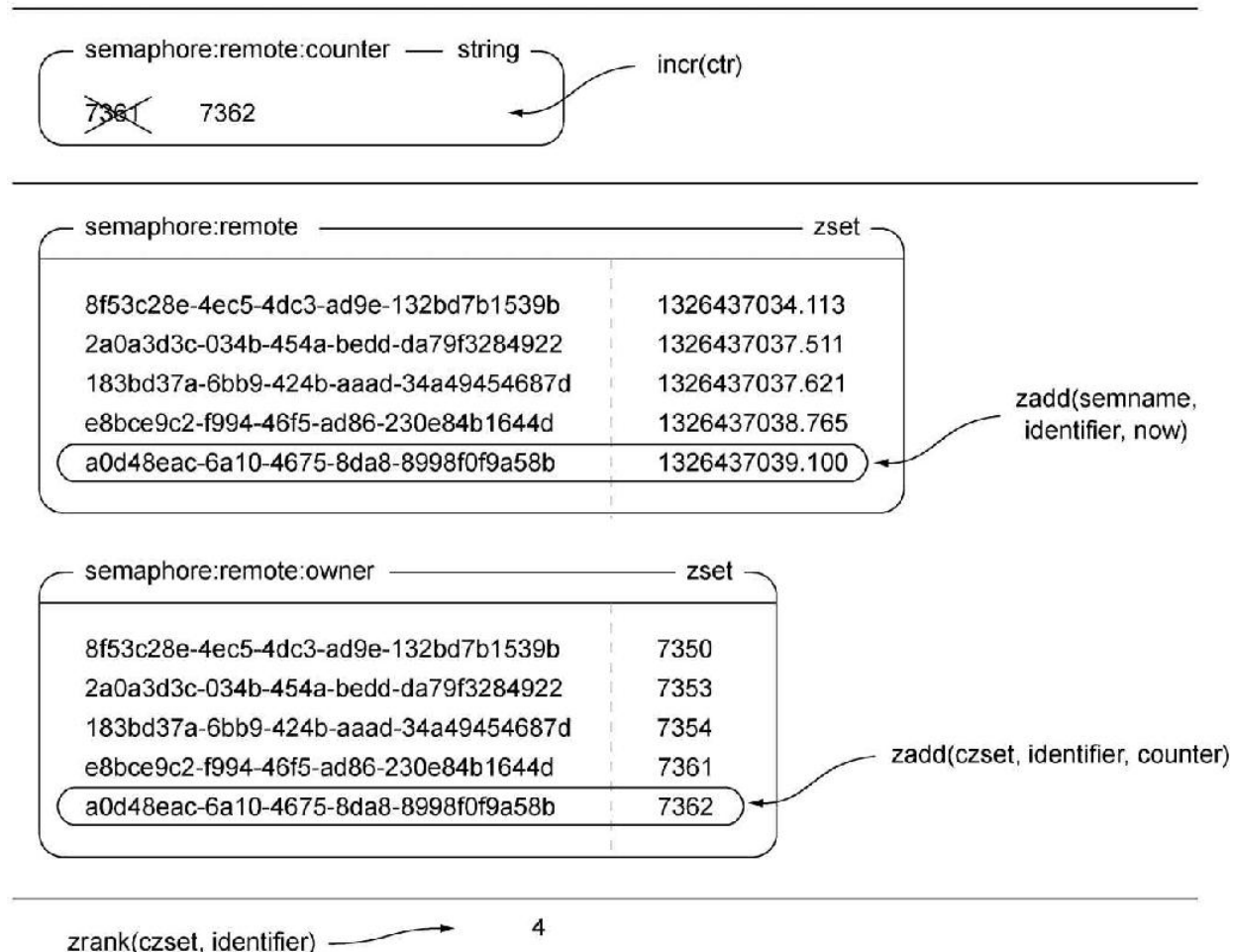
ID 和计数器值

64 位的计数器值

6-8 位的 ID 8372 1326437039.100

5

因为没有超时信号量存在，所以zremrangebyscore()和zinterstore()两个调用没有执行任何动作。



6-8 acquire_fair_semaphore()

该函数用于获取公平信号量。该函数首先会检查信号量是否存在，如果不存在，则会创建一个新的信号量。然后，它会检查信号量的当前值是否大于0。如果是，则会将信号量的值减1，并返回True。如果不是，则会将信号量的值加1，并返回False。该函数的实现如下：

6-15 release_fair_semaphore()

```
def release_fair_semaphore(conn, semname, identifier):
    pipeline = conn.pipeline(True)
    pipeline.zrem(semname, identifier)
    pipeline.zrem(semname + ':owner', identifier)
    return pipeline.execute()[0]
```

返回 True 表示信号量已被正确地释放，返回 False 则表示想要释放的信号量已经因为超时而被删除了。

更新客户端持有的信号量。

告知调用者，客户端已经失去了信号量。

```
def refresh_fair_semaphore(conn, semname, identifier):
    if conn.zadd(semname, identifier, time.time()):
        release_fair_semaphore(conn, semname, identifier)
        return False
    return True
```

← 客户端仍然持有信号量。

```
def refresh_fair_semaphore(conn, semname, identifier):
    if conn.zadd(semname, identifier, time.time()):
        release_fair_semaphore(conn, semname, identifier)
        return False
    return True
```

```
def refresh_fair_semaphore(conn, semname, identifier):
    if conn.zadd(semname, identifier, time.time()):
        release_fair_semaphore(conn, semname, identifier)
        return False
    return True
```

6.3.4 互斥锁

6.2 互斥锁

A B A B

——

redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

6.2.5 Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 6-17 acquire_semaphore_with_lock()

```
def acquire_semaphore_with_lock(conn, semname, limit, timeout=10):
    identifier = acquire_lock(conn, semname, acquire_timeout=.01)
    if identifier:
        try:
            return acquire_fair_semaphore(conn, semname, limit, timeout)
        finally:
            release_lock(conn, semname, identifier)
```

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

- Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。
- Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。
- Redis 的 semaphore 功能，在 Redis 2.8.12 版本中引入，用于实现分布式锁。

6.2

API

robots.txt
3

ePUBw.COM ePUBw.COM

6.4 ☐☐☐☐

Web
task queue
ActiveMQ
RabbitMQ
Gearman
Amazon SQS

[illegible]

6.4.1 □□□□□□

队列：先进先出（FIFO）
 栈：后进先出（LIFO）
 堆：无序的
 优先级队列：有序的

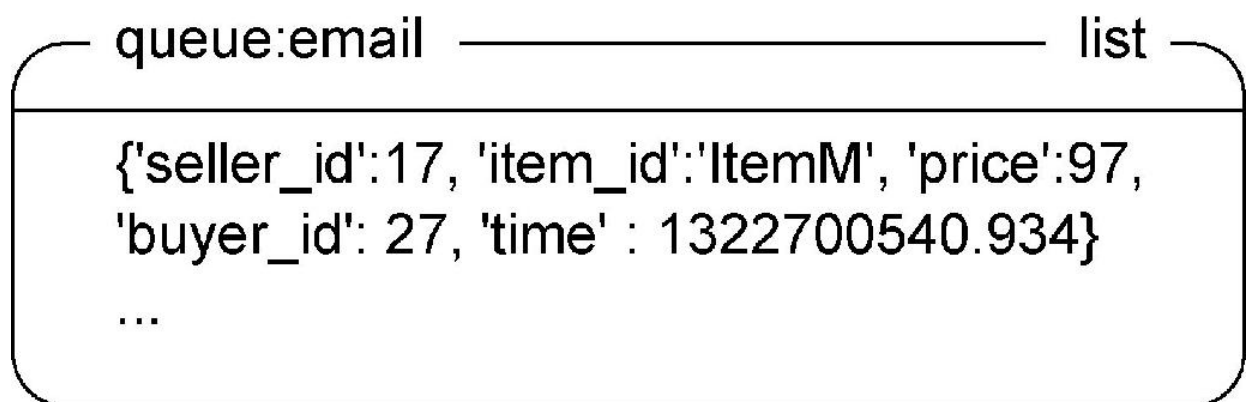
Fake Game

Fake Game

code flow
worker process

“”first-comefirst-served
35Redis
RPUSHLPUSHRPOPLOP
RPUSH
BLPOP30
30

RedisJSON6-9



6-9

将待发送邮件的字典数据序列化成为JSON字符串
通过conn.rpush()方法将JSON字符串推入队列
返回JSON字符串
图6-18 发送邮件队列

图6-18 send_sold_email_via_queue()

```
def send_sold_email_via_queue(conn, seller, item, price, buyer):  
    data = {  
        'seller_id': seller,  
        'item_id': item,  
        'price': price,  
        'buyer_id': buyer,  
        'time': time.time()  
    }  
    conn.rpush('queue:email', json.dumps(data))
```

准备好待发送邮件。
将待发送邮件推入队列里面。

send_sold_email_via_queue()函数
发送邮件

图6-19 发送邮件队列
BLPOP方法从队列中取出JSON字符串
返回JSON字符串

图6-19 process_sold_email_queue()

```
def process_sold_email_queue(conn):
    while not QUIT:
        packed = conn.blpop(['queue:email'], 30)
        if not packed:
            continue
        to_send = json.loads(packed[1])
        try:
            fetch_data_and_send_sold_email(to_send)
        except EmailSendError as err:
            log_error("Failed to send sold email", err, to_send)
        else:
            log_success("Sent sold email", to_send)
```

← 尝试获取一封待发送邮件。

← 队列里面暂时还没有待发送邮件，重试。

← 从 JSON 对象中解码出邮件信息。

← 从 JSON 对象中解码出邮件信息。

`process_sold_email_queue()` 函数

该函数接收一个连接对象 `conn` 作为参数，该连接对象是之前通过 `redis.Redis` 类实例化得到的。该函数会进入一个无限循环，直到接收到 `QUIT` 信号为止。在循环中，它会调用 `conn.blpop()` 方法，从 `queue:email` 队列中弹出一个元素。如果队列为空，则跳过该元素并继续下一次循环。如果队列不为空，则从弹出的元素中提取出邮件信息，并调用 `fetch_data_and_send_sold_email()` 函数。如果该函数抛出 `EmailSendError` 异常，则调用 `log_error()` 函数记录错误信息。否则，调用 `log_success()` 函数记录成功信息。

1. 函数定义

该函数使用 `BLPOP` 命令从队列中弹出一个元素。该命令的语法如下：

```
BLPOP queue timeout
```

其中，`queue` 是队列的名称，`timeout` 是等待的时间（以秒为单位）。该命令会返回一个列表，其中包含从队列中弹出的元素。如果队列为空，则返回 `None`。

在代码中，我们使用 `conn.blpop()` 方法调用该命令。该方法返回一个列表，其中包含从队列中弹出的元素。如果队列为空，则返回 `None`。

在代码中，我们使用 `json.loads()` 方法将弹出的元素解码为 JSON 对象。该方法的语法如下：

```
json.loads(json_string)
```

其中，`json_string` 是 JSON 字符串。该方法会返回一个 Python 对象，该对象是 JSON 字符串的 Python 表示形式。

在代码中，我们使用 `['FUNCTION_NAME', [ARG1, ARG2, ...]]` 格式来调用函数。

图 6-20 `worker_watch_queue()` 函数

```
def worker_watch_queue(conn, queue, callbacks):
    while not QUIT:
        packed = conn.blpop([queue], 30)
        if not packed:
            continue
        name, args = json.loads(packed[1])
        if name not in callbacks:
            log_error("Unknown callback %s"%name)
            continue
        callbacks[name](*args)
```

尝试从队列里面取出一项待执行任务。

队列为空，没有任务需要执行；重试。

解码任务信息。

没有找到任务指定的回调函数，用日志记录错误并重试。

执行任务。

1. 在 worker 进程中，使用 `blpop` 命令从队列中取出任务。

2. 解析任务信息

在 worker 进程中，使用 `json.loads` 命令解析任务信息。

任务信息包含任务名称和任务参数。

任务名称为 `BLPOP`，任务参数为 `BRPOP`。

任务名称为 `BLPOP`，任务参数为 `BRPOP`。

在 worker 进程中，使用 `3` 秒的超时时间。

在 worker 进程中，使用 `6-20` 秒的超时时间。

`worker_watch_queue()` 函数用于从队列中取出任务并执行。

在 `6-21` 图中。

图 6-21 worker_watch_queues() 函数

- 在 Redis 中，使用 `while` 循环来等待数据可用。在 Redis 中，使用 `while` 循环来等待数据可用。
- 在 Redis 中，使用 `while` 循环来等待数据可用。在 Redis 中，使用 `while` 循环来等待数据可用。
- 在 Redis 中，使用 `while` 循环来等待数据可用。在 Redis 中，使用 `while` 循环来等待数据可用。

在 Redis 中，使用 `while` 循环来等待数据可用。在 Redis 中，使用 `while` 循环来等待数据可用。

在 Redis 中，使用 `while` 循环来等待数据可用。在 Redis 中，使用 `while` 循环来等待数据可用。

在 Redis 中，使用 `while` 循环来等待数据可用。在 Redis 中，使用 `while` 循环来等待数据可用。

在 Redis 中，使用 `ZSET queue` 来等待数据可用。在 Redis 中，使用 `ZSET queue` 来等待数据可用。

在 Redis 中，使用 `ZSET queue` 来等待数据可用。在 Redis 中，使用 `ZSET queue` 来等待数据可用。

在 Redis 中，使用 `ZSET queue` 来等待数据可用。在 Redis 中，使用 `ZSET queue` 来等待数据可用。

在 Redis 中，使用 `ZSET queue` 来等待数据可用。在 Redis 中，使用 `ZSET queue` 来等待数据可用。

在 Redis 中，使用 `ZSET queue` 来等待数据可用。在 Redis 中，使用 `ZSET queue` 来等待数据可用。

在 Redis 中，使用 `ZSET queue` 来等待数据可用。在 Redis 中，使用 `ZSET queue` 来等待数据可用。

6-22 execute_later()

```

def execute_later(conn, queue, name, args, delay=0):
    identifier = str(uuid.uuid4())
    item = json.dumps([identifier, queue, name, args])
    if delay > 0:
        conn.zadd('delayed:', item, time.time() + delay)
    else:
        conn.rpush('queue:' + queue, item)
    return identifier

```

execute_later() 6-10

delayed:	zset
'["886148f7-...", "medium", "send_sold_email", [...]]'	1331850212.365
'["6c66e812-...", "medium", "send_sold_email", [...]]'	1332282209.459

□6-10 □□□□□□□□□□□□□□

Redis 是运行在 UNIX 上的
Redis 是运行在 UNIX 上的
Redis 是运行在 UNIX 上的
Redis 是运行在 UNIX 上的
Redis 是运行在 UNIX 上的
Redis 是运行在 UNIX 上的

redis 6.2.3 poll_queue() 实现

6-23 poll_queue() 实现

6-23 poll_queue() 实现

```
def poll_queue(conn):  
    while not QUIT:  
        item = conn.zrange('delayed:', 0, 0, withscores=True)  # 获取队列中的  
                                                                    # 第一个任务。  
        if not item or item[0][1] > time.time():  
            time.sleep(.01)  # 队列没有包含任何任务, 或者  
                               # 任务的执行时间未到。  
            continue  
        item = item[0][0]  
        identifier, queue, function, args = json.loads(item)  
        locked = acquire_lock(conn, identifier)  # 为了对任务进行移动,  
        if not locked:  # 尝试获取锁。  
            continue  
        if conn.zrem('delayed:', item):  
            conn.rpush('queue:' + queue, item)  # 将任务推入适当的任务  
                                                    # 队列里面。  
        release_lock(conn, identifier, locked)  # 释放锁。
```

6-23 poll_queue() 实现

redis 6.2.3 poll_queue() 实现

redis 6.2.3 poll_queue() 实现

redis 6.2.3 poll_queue() 实现

redis 6.2.3 poll_queue() 实现

100 redis 6.2.3 poll_queue() 实现

redis 6.2.3

redis 6.2.3 poll_queue() 实现

redis 6.2.3 poll_queue() 实现

redis 6.2.3

在代码中，我们使用了一个字典来映射不同的优先级。字典的键是优先级的名称，值是它们的延迟时间。我们使用字典的get方法来获取每个优先级的延迟时间。然后，我们使用字典的items方法来获取字典中的所有项。最后，我们使用sorted函数来对字典中的项进行排序。排序的依据是字典的值（延迟时间）。排序后的字典项被存储在worker_watch_queues列表中。最后，我们使用worker_watch_queues方法来访问这个列表。

在代码中，我们使用了一个字典来映射不同的优先级。字典的键是优先级的名称，值是它们的延迟时间。我们使用字典的get方法来获取每个优先级的延迟时间。然后，我们使用字典的items方法来获取字典中的所有项。最后，我们使用sorted函数来对字典中的项进行排序。排序的依据是字典的值（延迟时间）。排序后的字典项被存储在worker_watch_queues列表中。最后，我们使用worker_watch_queues方法来访问这个列表。

在代码中，我们使用了一个字典来映射不同的优先级。字典的键是优先级的名称，值是它们的延迟时间。我们使用字典的get方法来获取每个优先级的延迟时间。然后，我们使用字典的items方法来获取字典中的所有项。最后，我们使用sorted函数来对字典中的项进行排序。排序的依据是字典的值（延迟时间）。排序后的字典项被存储在worker_watch_queues列表中。最后，我们使用worker_watch_queues方法来访问这个列表。

在代码中，我们使用了一个字典来映射不同的优先级。字典的键是优先级的名称，值是它们的延迟时间。我们使用字典的get方法来获取每个优先级的延迟时间。然后，我们使用字典的items方法来获取字典中的所有项。最后，我们使用sorted函数来对字典中的项进行排序。排序的依据是字典的值（延迟时间）。排序后的字典项被存储在worker_watch_queues列表中。最后，我们使用worker_watch_queues方法来访问这个列表。

在代码中，我们使用了一个字典来映射不同的优先级。字典的键是优先级的名称，值是它们的延迟时间。我们使用字典的get方法来获取每个优先级的延迟时间。然后，我们使用字典的items方法来获取字典中的所有项。最后，我们使用sorted函数来对字典中的项进行排序。排序的依据是字典的值（延迟时间）。排序后的字典项被存储在worker_watch_queues列表中。最后，我们使用worker_watch_queues方法来访问这个列表。

在代码中，我们使用了一个字典来映射不同的优先级。字典的键是优先级的名称，值是它们的延迟时间。我们使用字典的get方法来获取每个优先级的延迟时间。然后，我们使用字典的items方法来获取字典中的所有项。最后，我们使用sorted函数来对字典中的项进行排序。排序的依据是字典的值（延迟时间）。排序后的字典项被存储在worker_watch_queues列表中。最后，我们使用worker_watch_queues方法来访问这个列表。

6.5 消息队列

消息队列（Message Queue，MQ）是分布式系统中用于在发送者（producer）和接收者（consumer）之间传递消息的中间件。它通常用于解耦系统组件，提高系统的可扩展性和可靠性。消息队列可以分为两种类型：push messaging 和 pull messaging。

push messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

pull messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

消息队列（Message Queue，MQ）是分布式系统中用于在发送者（producer）和接收者（consumer）之间传递消息的中间件。它通常用于解耦系统组件，提高系统的可扩展性和可靠性。消息队列可以分为两种类型：push messaging 和 pull messaging。

push messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

pull messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

消息队列（Message Queue，MQ）是分布式系统中用于在发送者（producer）和接收者（consumer）之间传递消息的中间件。它通常用于解耦系统组件，提高系统的可扩展性和可靠性。消息队列可以分为两种类型：push messaging 和 pull messaging。

push messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

pull messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

消息队列（Message Queue，MQ）是分布式系统中用于在发送者（producer）和接收者（consumer）之间传递消息的中间件。它通常用于解耦系统组件，提高系统的可扩展性和可靠性。消息队列可以分为两种类型：push messaging 和 pull messaging。

push messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

pull messaging 是指生产者（producer）将消息推送到消息队列（如 Redis）中，消费者（consumer）从队列中拉取消息。这种模式通常使用 PUBLISH 和 SUBSCRIBE 命令。Redis 支持最多 3 个消费者同时从同一个队列中拉取消息。

6.5.1 消息队列的部署和配置

Redis 消息队列的部署和配置相对简单。首先，需要安装 Redis 并配置好网络。然后，需要安装 Redis 客户端（如 Redis CLI）并配置好连接。最后，需要安装 Redis 消息队列插件（如 RedisMQ）并配置好消息队列。Redis 消息队列插件支持 PUBLISH 和 SUBSCRIBE 命令，可以用于实现消息队列功能。

Redis 消息队列的部署和配置相对简单。首先，需要安装 Redis 并配置好网络。然后，需要安装 Redis 客户端（如 Redis CLI）并配置好连接。最后，需要安装 Redis 消息队列插件（如 RedisMQ）并配置好消息队列。Redis 消息队列插件支持 PUBLISH 和 SUBSCRIBE 命令，可以用于实现消息队列功能。

```

    Fake Game Fake Garage
    Redis Web
    Redis

```

[illegible]

6.4.1
6-11
jack451

```
— mailbox:jack451 ————— list —
{'sender':'jill84', 'msg':'Are you coming or not?', 'ts':133066...}
{'sender':'mom65', 'msg':'Did you hear about aunt Elly?', ...}
```

```

6-11  jack451XXXXXXXXXXJillXXXXXXXXjack451XXXXX

```

PUBLISH SUBSCRIBE

Redis 的发布/订阅功能，Redis 的发布/订阅功能，Redis 的发布/订阅功能

PUBLISH SUBSCRIBE

6.5.2

PUBLISH SUBSCRIBE Redis PUBLISH SUBSCRIBE group chat “ ”

Fake Garage Fake Garage PUBLISH SUBSCRIBE

ID 6-12 ID

chat:827 zset	
jason22	5
jeff24	6

seen:jason22 zset	
827	5
729	10

chat:729 zset	
michelle19	10
jason22	10
jenny530	11

seen:jeff24 zset	
827	6

图6-12 使用Redis的ZSET数据类型存储chat和seen信息。chat ZSET的键是chat ID，值是用户ID。seen ZSET的键是用户ID，值是chat ID。

图6-12展示了jason22和jeff24在chat:827中的信息。jason22的seen ZSET中只有827，而jeff24的seen ZSET中只有827。

1. 创建聊天室

在创建聊天室时，我们需要将聊天室的ID和创建时间存储在Redis中。我们使用ZSET数据类型来存储聊天室的信息。每个聊天室的键是chat ID，值是创建时间。我们使用0作为初始值，表示聊天室刚刚创建。图6-24展示了创建聊天室的函数create_chat()。

图6-24 create_chat()函数

```
def create_chat(conn, sender, recipients, message, chat_id=None):
    chat_id = chat_id or str(conn.incr('ids:chat:'))
    recipients.append(sender)
    recipientsd = dict((r, 0) for r in recipients)
    pipeline = conn.pipeline(True)
    pipeline.zadd('chat:' + chat_id, **recipientsd)
    for rec in recipients:
        pipeline.zadd('seen:' + rec, chat_id, 0)
    pipeline.execute()
    return send_message(conn, chat_id, sender, message)
```

← 获得新的群组 ID。

创建一个由用户和分值组成的字典，字典里面的信息将被添加到有序集合里面。

← 将所有参与群聊的用户添加到有序集合里面。

初始化已读有序集合。

← 发送消息。

`create_chat()` 使用 `dict()` 来创建字典

`generator expression` 来创建字典

`ZADD` 来添加数据

在 Redis 中，有序集合的键名和成员名都是字符串

Python 中，有序集合的键名和成员名都是字符串

在 Redis 中，有序集合的键名和成员名都是字符串

在 Redis 中，有序集合的键名和成员名都是字符串

2. 有序集合

在 Redis 中，有序集合的键名和成员名都是字符串

在 Redis 中，有序集合的键名和成员名都是字符串

在 Redis 中，有序集合的键名和成员名都是字符串

在 Redis 中，有序集合的键名和成员名都是字符串

```
def send_message(conn, chat_id, sender, message):
    identifier = acquire_lock(conn, 'chat:' + chat_id)
    if not identifier:
        raise Exception("Couldn't get the lock")
    try:
        mid = conn.incr('ids:' + chat_id)
        ts = time.time()
        packed = json.dumps({
            'id': mid,
            'ts': ts,
            'sender': sender,
            'message': message,
        })

        conn.zadd('msgs:' + chat_id, packed, mid)
    finally:
        release_lock(conn, 'chat:' + chat_id, identifier)
    return chat_id
```

筹备待发送
的消息。

← 将消息发送至
群组。

在 Redis 中，我们使用 `send_message()` 函数来发送消息。该函数使用 `incr` 来生成消息 ID，使用 `zadd` 将消息添加到 Redis 的有序集合中。我们使用 `WATCH` 和 `MULTI` 来确保消息发送的原子性。最后，我们使用 `EXEC` 来执行事务。

在 Redis 中，我们使用 `send_message()` 函数来发送消息。该函数使用 `incr` 来生成消息 ID，使用 `zadd` 将消息添加到 Redis 的有序集合中。我们使用 `WATCH` 和 `MULTI` 来确保消息发送的原子性。最后，我们使用 `EXEC` 来执行事务。

3. 排序

在 Redis 中，我们使用 `ZRANGE` 函数来返回有序集合中的元素。该函数接受两个参数：起始 ID 和结束 ID。我们使用 `ZRANGEBYSCORE` 函数来返回有序集合中的元素，该函数接受两个参数：起始分数和结束分数。

ID

6-26

6-26 fetch_pending_messages()

```
def fetch_pending_messages(conn, recipient):
    seen = conn.zrange('seen:' + recipient, 0, -1, withscores=True)
    pipeline = conn.pipeline(True)

    for chat_id, seen_id in seen:
        pipeline.zrangebyscore(
            'msgs:' + chat_id, seen_id+1, 'inf')
        chat_info = zip(seen, pipeline.execute())
    for i, ((chat_id, seen_id), messages) in enumerate(chat_info):
        if not messages:
            continue
        messages[:] = map(json.loads, messages)
        seen_id = messages[-1]['id']
        conn.zadd('chat:' + chat_id, recipient, seen_id)

        min_id = conn.zrange(
            'chat:' + chat_id, 0, 0, withscores=True)

        pipeline.zadd('seen:' + recipient, chat_id, seen_id)
        if min_id:
            pipeline.zremrangebyscore(
                'msgs:' + chat_id, 0, min_id[0][1])
        chat_info[i] = (chat_id, messages)
    pipeline.execute()
    return chat_info
```

获取最后接收到的消息的 ID。

获取所有未读消息。

这些数据将被返回给函数调用者。

使用最新收到的消息来更新群组有序集合。

找出那些所有人都已经阅读过的消息。

更新已读消息有序集合。

清除那些已经被所有人阅读过的消息。

4

ID

返回消息ID列表

返回消息ID列表6-27

6-27 join_chat()

```
def join_chat(conn, chat_id, user):
    message_id = int(conn.get('ids:' + chat_id))
    pipeline = conn.pipeline(True)
    pipeline.zadd('chat:' + chat_id, user, message_id)
    pipeline.zadd('seen:' + user, chat_id, message_id)
    pipeline.execute()
```

将用户添加到群组成员列表里面。

取得最新群组消息的ID。

将群组添加到用户的已读列表里面。

join_chat()

返回消息ID列表

返回消息ID列表6-28

6-28 leave_chat()

```
def leave_chat(conn, chat_id, user):
    pipeline = conn.pipeline(True)
    pipeline.zrem('chat:' + chat_id, user)
    pipeline.zrem('seen:' + user, chat_id)
    pipeline.zcard('chat:' + chat_id)
    if not pipeline.execute()[-1]:
        pipeline.delete('msgs:' + chat_id)
        pipeline.delete('ids:' + chat_id)
        pipeline.execute()
    else:
        oldest = conn.zrange('chat:' + chat_id, 0, 0, withscores=True)
        conn.zremrangebyscore('msgs:' + chat_id, 0, oldest[0][1])
```

查找群组剩余成员的数量。

从群组里面移除给定的用户。

删除群组。

找出那些已经被所有成员阅读过的消息。

删除那些已经被所有成员阅读过的消息。

6.6 Redis 数据库

Redis 是一个开源的、基于内存的、支持持久化的数据库。它支持多种数据类型，如字符串、列表、集合、有序集合、哈希等。Redis 还支持主从复制、哨兵、集群等功能。Redis 的持久化方式有两种：RDB 和 AOF。RDB 是 Redis 的默认持久化方式，它会将数据库中的数据快照保存到磁盘上。AOF 则是将 Redis 的每条写操作都记录到日志文件中。Redis 还支持多种客户端，如 Redis CLI、Redis Python 客户端、Redis Java 客户端等。Redis 的官方网站是 <http://redis.io>。

Redis 的持久化方式有两种：RDB 和 AOF。RDB 是 Redis 的默认持久化方式，它会将数据库中的数据快照保存到磁盘上。AOF 则是将 Redis 的每条写操作都记录到日志文件中。Redis 还支持多种客户端，如 Redis CLI、Redis Python 客户端、Redis Java 客户端等。Redis 的官方网站是 <http://redis.io>。

Redis 的持久化方式有两种：RDB 和 AOF。RDB 是 Redis 的默认持久化方式，它会将数据库中的数据快照保存到磁盘上。AOF 则是将 Redis 的每条写操作都记录到日志文件中。Redis 还支持多种客户端，如 Redis CLI、Redis Python 客户端、Redis Java 客户端等。Redis 的官方网站是 <http://redis.io>。

6.6.1 网络攻击与防御

攻击者5通过IP地址扫描发现目标主机存在Fake Game漏洞，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。

攻击者5通过Fake Game漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过10个攻击点攻击目标主机，攻击者5通过73个攻击点攻击目标主机。攻击者5通过IP地址扫描发现目标主机存在Fake Game漏洞，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。

攻击者5通过MapReduce^④攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过MapReduce攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过IP地址扫描发现目标主机存在Fake Game漏洞，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过lookup table攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过Python攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过20个攻击点攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过Redis攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过MapReduce攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过Redis攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过MapReduce攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。

攻击者5通过NFS、Samba攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。攻击者5通过Fake Game攻击目标主机，攻击者通过该漏洞获取目标主机的GB信息，攻击者通过该信息进一步攻击目标主机。

Redis 1000
Redis 1000
Redis 300
Redis 300
Redis

10 10
Redis
350 000 10% 90%
Redis

5.3 IP
IP
Redis

173.194.38.137 2011-10-10 13:55:36 achievement-762

Redis 6-29

6-29

```

        aggregates = defaultdict(lambda: defaultdict(int))
    def daily_country_aggregate(conn, line):
        if line:
            line = line.split()
            ip = line[0]
            day = line[1]
            country = find_city_by_ip_local(ip)[2]
            aggregates[day][country] += 1
            return
        for day, aggregate in aggregates.items():
            conn.zadd('daily:country:' + day, **aggregate)
            del aggregates[day]

```

提取日志行中的信息。

对本地聚合数据执行自增操作。

准备本地聚合数据字典。

根据 IP 地址判断用户所在国家。

当天的日志文件已经处理完毕，将聚合计算的结果写入 Redis 里面。

daily_country_aggregate() 函数负责从日志文件中提取信息，并对本地聚合数据执行自增操作。当处理完当天的日志文件后，将聚合计算的结果写入 Redis 数据库。

6.6.2 数据聚合

在本节中，我们将介绍如何使用 Redis 的 ZSET 数据类型来实现数据聚合。在 6.5.2 节中，我们介绍了如何使用 Redis 的 ZSET 数据类型来实现数据聚合。在本节中，我们将介绍如何使用 Redis 的 ZSET 数据类型来实现数据聚合。在本节中，我们将介绍如何使用 Redis 的 ZSET 数据类型来实现数据聚合。

6-30 copy_logs_to_redis()

```

def copy_logs_to_redis(conn, path, channel, count=10,
                      limit=2**30, quit_when_done=True):
    bytes_in_redis = 0
    waiting = deque()
    create_chat(conn, 'source', map(str, range(count)), '', channel)
    count = str(count)
    for logfile in sorted(os.listdir(path)):
        full_path = os.path.join(path, logfile)

        fsize = os.stat(full_path).st_size
        while bytes_in_redis + fsize > limit:
            cleaned = _clean(conn, channel, waiting, count)
            if cleaned:
                bytes_in_redis -= cleaned
            else:
                time.sleep(.25)

        with open(full_path, 'rb') as inp:
            block = ''
            while block:
                block = inp.read(2**17)
                conn.append(channel+logfile, block)

        send_message(conn, channel, 'source', logfile)

        bytes_in_redis += fsize
        waiting.append((logfile, fsize))

    if quit_when_done:
        send_message(conn, channel, 'source', ':done')

    while waiting:
        cleaned = _clean(conn, channel, waiting, count)
        if cleaned:
            bytes_in_redis -= cleaned
        else:
            time.sleep(.25)

    def _clean(conn, channel, waiting, count):
        if not waiting:
            return 0
        w0 = waiting[0][0]
        if conn.get(channel + w0 + ':done') == count:
            conn.delete(channel + w0, channel + w0 + ':done')
            return waiting.popleft()[1]
        return 0

```

创建用于向客户端发送消息的群组。

遍历所有日志文件。

如果程序需要更多空间，那么清除已经处理完毕的文件。

将文件上传至 Redis。

对本地记录的 Redis 内存占用量相关信息进行更新。

所有日志文件已经处理完毕，向监听者报告此事。

在工作完成之后，清理无用的日志文件。

对 Redis 进行清理的详细步骤。

提醒监听者，文件已经准备就绪。

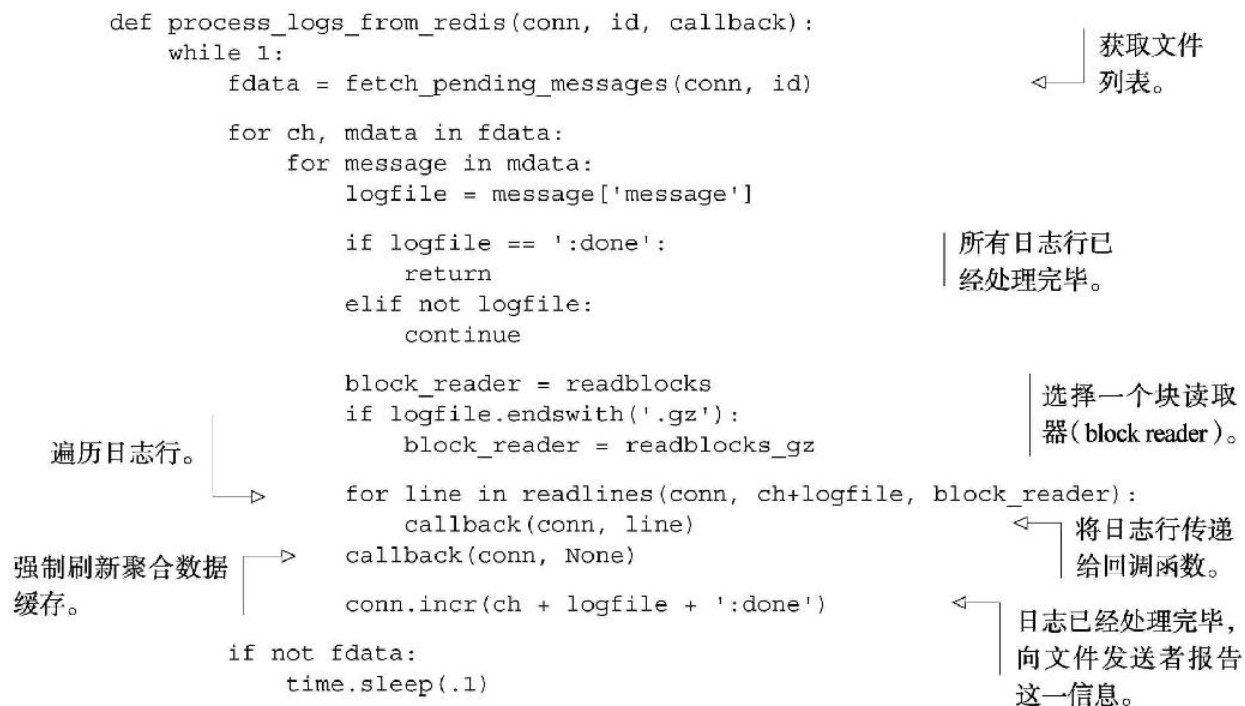
Redis copy_logs_to_redis()

Redis

6.6.3 日志处理

Redis 日志文件位于 `/var/log/redis/redis.log`。Redis 日志文件默认是追加模式，因此日志文件会越来越大。图 6-31 展示了 Redis 日志处理流程。

图 6-31 `process_logs_from_redis()` 函数



Redis 日志处理流程如下：Redis 日志文件位于 `/var/log/redis/redis.log`。Redis 日志文件默认是追加模式，因此日志文件会越来越大。图 6-31 展示了 Redis 日志处理流程。

6.6.4 日志处理

解码decode
 6-32 readlines() Redis
 block iterating callback
 line break
 Python rfind()
 split()

6-32 readlines()

```

def readlines(conn, key, rblocks):
    out = ''
    for block in rblocks(conn, key):
        out += block
        posn = out.rfind('\n')
        if posn >= 0:
            for line in out[:posn].split('\n'):
                yield line + '\n'
            out = out[posn+1:]
    if not block:
        yield out
        break
    
```

找到一个换行符。 →
 根据换行符来分割日志行。 →
 所有数据块已经处理完毕。 →
 查找位于文本最右端的换行符；如果换行符不存在，那么 rfind() 返回-1。
 向调用者返回每个行。
 保留余下的数据。

readlines()
 yield

yield 6-32 yield Python
 yield

Python

<http://mng.bz/Z2b1>

`readlines()` `readblocks()`
`readblocks_gz()` Redis `readblocks()`
`readblocks_gz()` gzip
6-33
`readblocks()`

6-33 `readblocks()`

```
def readblocks(conn, key, blocksize=2**17):  
    lb = blocksize  
    pos = 0  
    while lb == blocksize:  
        block = conn.substr(key, pos, pos + blocksize - 1)  
        yield block  
        lb = len(block)  
        pos += lb  
    yield ''
```

尽可能地读取更多数据，
直到出现不完整读操作
(partial read) 为止。

为下一次遍历
做准备。

`readblocks()`
—— memcached
Redis gzip
6-34
`readblocks_gz()`

6-34 `readblocks_gz()`

```

def readblocks_gz(conn, key):
    inp = ''
    decoder = None
    for block in readblocks(conn, key, 2**17):
        if not decoder:
            inp += block
            try:
                if inp[:3] != "\x1f\x8b\x08":
                    raise IOError("invalid gzip data")
                i = 10
                flag = ord(inp[3])
                if flag & 4:
                    i += 2 + ord(inp[i]) + 256*ord(inp[i+1])
                if flag & 8:
                    i = inp.index('\0', i) + 1
                if flag & 16:
                    i = inp.index('\0', i) + 1
                if flag & 2:
                    i += 2

                if i > len(inp):
                    raise IndexError("not enough data")
            except (IndexError, ValueError):
                continue

        else:
            block = inp[i:]
            inp = None
            decoder = zlib.decompressobj(-zlib.MAX_WBITS)
            if not block:
                continue

    if not block:
        yield decoder.flush()
        break

    yield decoder.decompress(block)

```

从 Redis 里面
读入原始数据。

分析头信息以便取得被压缩
的数据。

程序读取的头信息
并不完整。

已经找到头信息，准备
好相应的解压程序。

所有数据已经处理完
毕，向调用者返回最
后剩下的数据块。

向调用者返回解
压后的数据块。

readblocks_gz() 支持 gzip 1/2 1/5 bzip2 lzma xz lz4 lzop snappy QuickLZ gz CPU

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

6.7 总结

本文介绍了 Redis 6 版本中新增的 `WATCH` 命令，以及 Redis 6.2 版本中新增的 `Redis` 命令。本文还介绍了 Redis 4.6 版本中新增的 `Redis` 命令。本文还介绍了 Redis 6.4.2 版本中新增的 `Redis` 命令。

本文介绍了 Redis 6 版本中新增的 `WATCH` 命令，以及 Redis 6.2 版本中新增的 `Redis` 命令。本文还介绍了 Redis 4.6 版本中新增的 `Redis` 命令。本文还介绍了 Redis 6.4.2 版本中新增的 `Redis` 命令。

本文介绍了 Redis 6 版本中新增的 `WATCH` 命令，以及 Redis 6.2 版本中新增的 `Redis` 命令。本文还介绍了 Redis 4.6 版本中新增的 `Redis` 命令。本文还介绍了 Redis 6.4.2 版本中新增的 `Redis` 命令。

本文介绍了 Redis 6 版本中新增的 `WATCH` 命令，以及 Redis 6.2 版本中新增的 `Redis` 命令。本文还介绍了 Redis 4.6 版本中新增的 `Redis` 命令。本文还介绍了 Redis 6.4.2 版本中新增的 `Redis` 命令。

① 2010 Facebook “Betty White
Facebook Betty White
Saturday Night Live SNL
——

② Redis 5
10

③ PUBLISH SUBSCRIBE
Redis

④ MapReduce Map/Reduce Google

ePUBw.COM ePUBw.COM

7

--	--	--	--	--	--

- Redis
-
-
-

Redis Redis
Redis Redis
search-based problem
Redis

```

Redis
Redis
ad-targeting engine

```

Redis

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

7.1 Redis

Redis is a key-value database. It is a simple database that stores data in memory. It is a fast database that can handle millions of requests per second. It is a distributed database that can be scaled horizontally. It is a database that can be used for a variety of applications, including caching, session storage, and real-time analytics. It is a database that is easy to use and integrate with other applications. It is a database that is highly available and fault-tolerant. It is a database that is secure and compliant with various regulations. It is a database that is open source and has a large community of users and developers. It is a database that is constantly evolving and improving. It is a database that is the future of data storage.

Redis is a key-value database. It is a simple database that stores data in memory. It is a fast database that can handle millions of requests per second. It is a distributed database that can be scaled horizontally. It is a database that can be used for a variety of applications, including caching, session storage, and real-time analytics. It is a database that is easy to use and integrate with other applications. It is a database that is highly available and fault-tolerant. It is a database that is secure and compliant with various regulations. It is a database that is open source and has a large community of users and developers. It is a database that is constantly evolving and improving. It is a database that is the future of data storage.

Redis is a key-value database. It is a simple database that stores data in memory. It is a fast database that can handle millions of requests per second. It is a distributed database that can be scaled horizontally. It is a database that can be used for a variety of applications, including caching, session storage, and real-time analytics. It is a database that is easy to use and integrate with other applications. It is a database that is highly available and fault-tolerant. It is a database that is secure and compliant with various regulations. It is a database that is open source and has a large community of users and developers. It is a database that is constantly evolving and improving. It is a database that is the future of data storage.

Redis is a key-value database. It is a simple database that stores data in memory. It is a fast database that can handle millions of requests per second. It is a distributed database that can be scaled horizontally. It is a database that can be used for a variety of applications, including caching, session storage, and real-time analytics. It is a database that is easy to use and integrate with other applications. It is a database that is highly available and fault-tolerant. It is a database that is secure and compliant with various regulations. It is a database that is open source and has a large community of users and developers. It is a database that is constantly evolving and improving. It is a database that is the future of data storage.

7.1.1 Redis

Redis is a key-value database. It is a simple database that stores data in memory. It is a fast database that can handle millions of requests per second. It is a distributed database that can be scaled horizontally. It is a database that can be used for a variety of applications, including caching, session storage, and real-time analytics. It is a database that is easy to use and integrate with other applications. It is a database that is highly available and fault-tolerant. It is a database that is secure and compliant with various regulations. It is a database that is open source and has a large community of users and developers. It is a database that is constantly evolving and improving. It is a database that is the future of data storage.

1. 在 Redis 中，使用 `hset` 命令可以设置哈希表中的键值对。
 2. 在 Redis 中，使用 `hget` 命令可以获取哈希表中的值。
 3. 在 Redis 中，使用 `hkeys` 命令可以获取哈希表中的所有键。
 4. 在 Redis 中，使用 `hvals` 命令可以获取哈希表中的所有值。
 5. 在 Redis 中，使用 `hdel` 命令可以删除哈希表中的键值对。
 6. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。
 7. 在 Redis 中，使用 `hincrby` 命令可以对哈希表中的值进行增量操作。
 8. 在 Redis 中，使用 `hincrbyfloat` 命令可以对哈希表中的值进行浮点增量操作。
 9. 在 Redis 中，使用 `hmset` 命令可以同时设置哈希表中的多个键值对。
 10. 在 Redis 中，使用 `hmget` 命令可以同时获取哈希表中的多个值。
 11. 在 Redis 中，使用 `hstrlen` 命令可以获取哈希表中值的字符串长度。
 12. 在 Redis 中，使用 `hexpire` 命令可以为哈希表中的键设置过期时间。
 13. 在 Redis 中，使用 `hexpireat` 命令可以为哈希表中的键设置具体的过期时间。
 14. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。
 15. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。

1. 在 Redis 中，使用 `hset` 命令可以设置哈希表中的键值对。
 2. 在 Redis 中，使用 `hget` 命令可以获取哈希表中的值。
 3. 在 Redis 中，使用 `hkeys` 命令可以获取哈希表中的所有键。
 4. 在 Redis 中，使用 `hvals` 命令可以获取哈希表中的所有值。
 5. 在 Redis 中，使用 `hdel` 命令可以删除哈希表中的键值对。
 6. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。
 7. 在 Redis 中，使用 `hincrby` 命令可以对哈希表中的值进行增量操作。
 8. 在 Redis 中，使用 `hincrbyfloat` 命令可以对哈希表中的值进行浮点增量操作。
 9. 在 Redis 中，使用 `hmset` 命令可以同时设置哈希表中的多个键值对。
 10. 在 Redis 中，使用 `hmget` 命令可以同时获取哈希表中的多个值。
 11. 在 Redis 中，使用 `hstrlen` 命令可以获取哈希表中值的字符串长度。
 12. 在 Redis 中，使用 `hexpire` 命令可以为哈希表中的键设置过期时间。
 13. 在 Redis 中，使用 `hexpireat` 命令可以为哈希表中的键设置具体的过期时间。
 14. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。
 15. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。

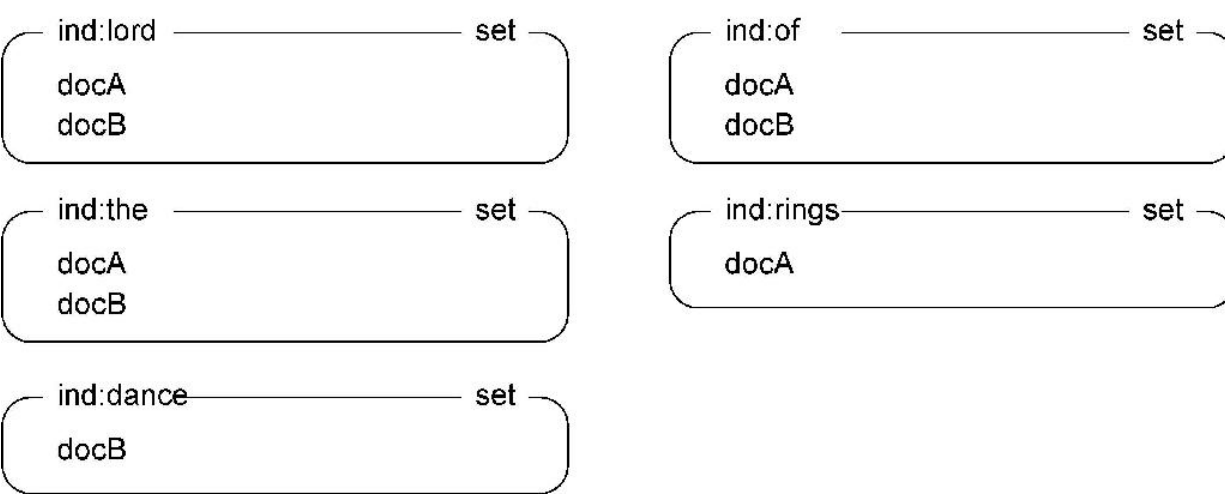


图 7-1 docA 和 docB 的 Redis 哈希表结构

1. 在 Redis 中，使用 `hset` 命令可以设置哈希表中的键值对。
 2. 在 Redis 中，使用 `hget` 命令可以获取哈希表中的值。
 3. 在 Redis 中，使用 `hkeys` 命令可以获取哈希表中的所有键。
 4. 在 Redis 中，使用 `hvals` 命令可以获取哈希表中的所有值。
 5. 在 Redis 中，使用 `hdel` 命令可以删除哈希表中的键值对。
 6. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。
 7. 在 Redis 中，使用 `hincrby` 命令可以对哈希表中的值进行增量操作。
 8. 在 Redis 中，使用 `hincrbyfloat` 命令可以对哈希表中的值进行浮点增量操作。
 9. 在 Redis 中，使用 `hmset` 命令可以同时设置哈希表中的多个键值对。
 10. 在 Redis 中，使用 `hmget` 命令可以同时获取哈希表中的多个值。
 11. 在 Redis 中，使用 `hstrlen` 命令可以获取哈希表中值的字符串长度。
 12. 在 Redis 中，使用 `hexpire` 命令可以为哈希表中的键设置过期时间。
 13. 在 Redis 中，使用 `hexpireat` 命令可以为哈希表中的键设置具体的过期时间。
 14. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。
 15. 在 Redis 中，使用 `hscan` 命令可以遍历哈希表。

1. 在 Redis 中，使用 `hset` 命令可以设置哈希表中的键值对。

原文：

In order to construct our SETs of documents, we must first examine our documents for words. The process of extracting words from documents is known as parsing and tokenization; we are producing a set of tokens (or words) that identify the document.

标记化

标记化之后的内容：

and are as construct document
documents examine extracting first
for from identify in is known must of
or order our parsing process
producing set sets that the to
tokenization tokens we words

移除非用词

移除非用词之后的内容：

construct document documents
examine extracting first identify
known order parsing process
producing set sets tokenization
tokens words

7-2 停用词和索引

在文本分析中，停用词（stop words）是指那些在文本中频繁出现但对语义贡献不大的词。这些词通常包括冠词、介词、连词、助词等。在索引构建过程中，停用词会被忽略，以减少索引的大小并提高检索效率。

7-1 从 <http://www.textfixer.com/resources/> 获取的非用词。

7-1 从 <http://www.textfixer.com/resources/> 获取的非用词。

```
STOP_WORDS = set('able about across after all almost also am among  
an and any are as at be because been but by can cannot could dear did  
do does either else ever every for from get got had has have he her  
hers him his how however if in into is it its just least let like  
likely may me might most must my neither no nor not of off often on  
only or other our own rather said say says she should since so some  
than that the their them then there these they this tis to too twas us  
wants was we were what when where which while who whom why will with  
would yet you your'.split())  
  
WORDS_RE = re.compile("[a-z']{2,}")  
  
def tokenize(content):  
    words = set()  
    for match in WORDS_RE.finditer(content.lower()):  
        word = match.group().strip("'")  
        if len(word) >= 2:  
            words.add(word)  
    return words - STOP_WORDS  
  
def index_document(conn, docid, content):  
    words = tokenize(content)  
    pipeline = conn.pipeline(True)  
    for word in words:  
        pipeline.sadd('idx:' + word, docid)  
    return len(pipeline.execute())
```

将文档中包含的单词存储到 Python 集合里面。

遍历文档中包含的所有单词。

返回一个集合，集合里面包含了所有被保留的、不是非用词的单词。

预先定义好从 <http://www.textfixer.com/resources/> 获取的非用词。

根据定义提取单词的正则表达式。

保留那些至少有两个字符长的单词。

剔除所有位于单词前面或后面的单引号。

对内容进行标记化处理，并取得处理产生的单词。

将文档添加到正确的反向索引集合里面。

计算一下，程序为这个文档添加了多少个独一无二的、不是非用词的单词。

of the 7-1

docA docB lord rings dance

lord of the rings dance

在 Redis 中，我们使用 `JSON.SET` 命令来设置 JSON 文档。该命令的语法如下：

```
JSON.SET key path json
```

其中，`key` 是键名，`path` 是文档中的路径，`json` 是 JSON 文档。我们使用 `index_document()` 方法来生成 JSON 文档。

在 Redis 中，我们使用 `JSON.GET` 命令来获取 JSON 文档。该命令的语法如下：

2. 集合操作

Redis 提供了丰富的集合操作命令，包括并集、交集、差集等。我们使用 `SINTER` 命令来计算两个集合的交集。该命令的语法如下：

```
SINTER key1 key2
```

其中，`key1` 和 `key2` 是两个集合的键名。我们使用 `SINTERSTORE` 命令来将交集结果存储到新的键中。该命令的语法如下：

```
SINTERSTORE destination key1 key2
```

其中，`destination` 是目标键名，`key1` 和 `key2` 是两个集合的键名。

在 Redis 中，我们使用 `SUNION` 命令来计算两个集合的并集。该命令的语法如下：

```
SUNION key1 key2
```

其中，`key1` 和 `key2` 是两个集合的键名。我们使用 `SUNIONSTORE` 命令来将并集结果存储到新的键中。该命令的语法如下：

```
SUNIONSTORE destination key1 key2
```

其中，`destination` 是目标键名，`key1` 和 `key2` 是两个集合的键名。

在 Redis 中，我们使用 `SUNION` 命令来计算两个集合的并集。该命令的语法如下：

```
SUNION key1 key2
```

其中，`key1` 和 `key2` 是两个集合的键名。我们使用 `SUNIONSTORE` 命令来将并集结果存储到新的键中。该命令的语法如下：

```
SUNIONSTORE destination key1 key2
```

其中，`destination` 是目标键名，`key1` 和 `key2` 是两个集合的键名。

Redis 的 SINTER、SUNION、SDIFF 命令。Redis 的 SINTER、SUNION、SDIFF 命令。

Redis 的 SINTER、SUNION、SDIFF 命令。Redis 的 SINTER、SUNION、SDIFF 命令。

7-2 Redis 的 SINTER、SUNION、SDIFF 命令

给每个单词加上 'idx:' 前缀。

设置事务流水线，确保每个调用都能获得一致的执行结果。

创建一个新的临时标识符。

吩咐 Redis 在将来自动删除这个集合。

为将要执行的集合操作设置相应的参数。

将结果集合的 ID 返回给调用者，以便做进一步的处理。

实际地执行操作。

执行交集计算的辅助函数。

执行并集计算的辅助函数。

执行差集计算的辅助函数。

```
def _set_common(conn, method, names, ttl=30, execute=True):
    id = str(uuid.uuid4())
    pipeline = conn.pipeline(True) if execute else conn
    names = ['idx:' + name for name in names]
    getattr(pipeline, method)('idx:' + id, *names)
    pipeline.expire('idx:' + id, ttl)
    if execute:
        pipeline.execute()
    return id

def intersect(conn, items, ttl=30, _execute=True):
    return _set_common(conn, 'sinterstore', items, ttl, _execute)

def union(conn, items, ttl=30, _execute=True):
    return _set_common(conn, 'sunionstore', items, ttl, _execute)

def difference(conn, items, ttl=30, _execute=True):
    return _set_common(conn, 'sdiffstore', items, ttl, _execute)
```

Redis 的 SINTER、SUNION、SDIFF 命令。Redis 的 SINTER、SUNION、SDIFF 命令。

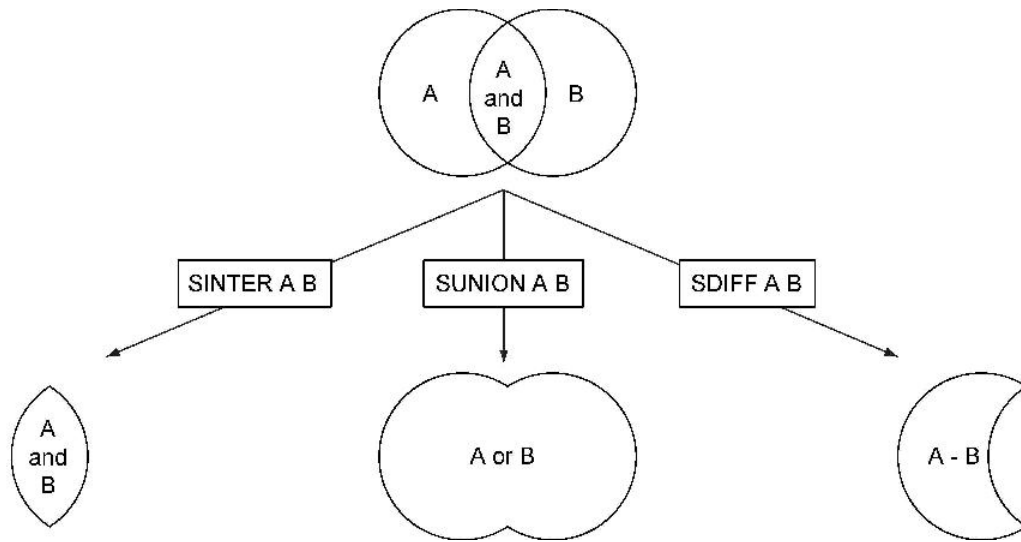


图7-3 集合的交集、并集和差集

集合的交集、并集和差集在Python中分别使用 `intersect()`、`union()` 和 `difference()` 方法实现。

例如：

3. 集合的交集

集合的交集是指两个集合中共同拥有的元素。在Python中，可以使用 `intersect()` 方法来实现。

例如，假设有两个集合 `A` 和 `B`，它们的交集可以通过以下代码计算：

```
set_A = {1, 2, 3, 4, 5}
set_B = {3, 4, 5, 6, 7}
intersection = set_A.intersection(set_B)
```

输出：

```
{3, 4, 5}
```

其中，`intersect()` 方法返回的是两个集合的交集，即 `{3, 4, 5}`。

除了使用 `intersect()` 方法外，还可以使用 `difference()` 方法来实现。

```
set_A = {1, 2, 3, 4, 5}
set_B = {3, 4, 5, 6, 7}
intersection = set_A.difference(set_B)
```

union() intersect()
+ - - -

7-3
Python

7-3

这个集合将用于存储不需要的单词。	QUERY_RE = re.compile("[+-]?[a-z']{2,}")	用于查找需要的单词、不需要的单词以及同义词的 正则表达式。
这个列表将用于存储需要执行交集计算的单词。	def parse(query): unwanted = set() all = [] current = set() for match in QUERY_RE.finditer(query.lower()): word = match.group() prefix = word[:1] if prefix in '+-': word = word[1:] else: prefix = None word = word.strip("'") if len(word) < 2 or word in STOP_WORDS: continue if prefix == '-': unwanted.add(word) continue if current and not prefix: all.append(list(current)) current = set() current.add(word) if current: all.append(list(current)) return all, list(unwanted)	这个集合将用于存储日 前已发现的同义词。 检查单词是否带有加号前缀或者减号前缀，如果有的话。 遍历搜索查询语句中的所有单词。
剔除所有位于单词前面或者后面的单引号，并略过所有非用词。		如果这是一个不需要的单词，那么将它添加到存储不需要单词的集合里面。
将正在处理的单词添加到同义词集合里面。		如果在同义词集合非空的情况下，遇到了一个不带+号前缀的单词，那么创建一个新的同义词集合。
		把所有剩余的单词都放到最后的交集计算里面进行处理。

connect connection disconnect disconnection


```
chat proxy
proxies

```

```
>>> parse(''
connect +connection +disconnect +disconnection
chat
-proxy -proxies'')
(['disconnection', 'connection', 'disconnect', 'connect'], ['chat'],
['proxies', 'proxy'])
>>>
```

connect disconnect chat
proxy proxies
—— parse_and_search()
parse() union()
intersect() difference()
7-4 parse_and_search()

7-4

```
def parse_and_search(conn, query, ttl=30):
    all, unwanted = parse(query)
    if not all:
        return None
```

 Δ

```
to_intersect = []
for syn in all:  ← 遍历各个同义词列表。
```

```

    to_intersect.append(union(conn, syn, ttl=ttl))
else:
    to_intersect.append(syn[0])

```

```
if len(to_intersect) > 1:  # 这个单词。
    intersect_result = intersect(conn, to_intersect, ttl=ttl)
```

```
if unwanted:
    unwanted.insert(0, intersect_result)
    return difference(conn, unwanted, ttl=ttl)
```

 Δ

```
parse_and_search() ID
```

```
00000000ID000000000000000000000000000000000000Fake Garage
```

```
index_document()
```

```
parse_and_search()
```

[illegible][illegible]

7.1.2 〇〇〇〇〇〇〇〇〇

[illegible][illegible][illegible]

3 Redis SORT
 Fake Garage
 ID 7-4

kb:doc276	hash
id	276
created	1324114412
updated	1327562777
title	Troubleshooting...
...	...

7-4

7-4 SORT
 parse_and_search()

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

7.2 集合

Redis 集合类型支持 `sort order` 参数，用于指定集合元素的排序方式。Redis 集合类型支持 `SORT` 命令。

Redis `SORT` 命令支持 `MAX` 参数，用于指定集合元素的排序方式。Redis `ZINTERSTORE` 命令支持 `MAX` 参数，用于指定集合元素的排序方式。

7.2.1 集合操作

Redis 集合类型支持 `ZINTERSTORE` 命令，用于计算两个集合的交集。Redis 集合类型支持 `ZUNIONSTORE` 命令，用于计算两个集合的并集。

Redis 集合类型支持 `SORT` 命令，用于对集合元素进行排序。Redis 集合类型支持 `SORT` 命令，用于对集合元素进行排序。

Redis 集合类型支持 `SORT` 命令，用于对集合元素进行排序。Redis 集合类型支持 `SORT` 命令，用于对集合元素进行排序。

尝试刷新
已有搜索
结果的生
存时间。

```
def search_and_zsort(conn, query, id=None, ttl=300, update=1, vote=0,
                    start=0, num=20, desc=True):
```

```
    if id and not conn.expire(id, ttl):
        id = None
```

```
    if not id:
        id = parse_and_search(conn, query, ttl=ttl)
```

```
        scored_search = {
            id: 0,
            'sort:update': update,
            'sort:votes': vote
        }
```

```
        id = zintersect(conn, scored_search, ttl)
```

```
    pipeline = conn.pipeline(True)
```

```
    pipeline.zcard('idx:' + id)
```

```
    if desc:
```

```
        pipeline.zrevrange('idx:' + id, start, start + num - 1)
```

```
    else:
```

```
        pipeline.zrange('idx:' + id, start, start + num - 1)
```

```
    results = pipeline.execute()
```

```
    return results[0], results[1], id
```

和之前一样，函数接受一个已有搜索结果的 ID 作为可选参数，以便在结果仍然可用的情况下，对其进行分页。

如果传入的结果已经过期，或者这是函数第一次进行搜索，那么执行标准的集合搜索操作。

函数在计算交集的时候也会用到传入的 ID 键，但这个键不会被用作排序权重 (weight)。

使用代码清单 7-7 定义的辅助函数执行交集计算。

获取结果有序集合的大小。

从搜索结果里面取出一页 (page)。

返回搜索结果，以及分页用的 ID 值。

对文章评分进行调整以平衡更新时间和投票数量。根据待排序数据的需要，投票数量可以被调整为 1、10、100，甚至更高。

search_and_zsort() 辅助函数 search_and_sort()

辅助函数

search_and_sort() 辅助函数 SORT

search_and_zsort() 辅助函数 ZINTERSTORE

辅助函数

search_and_zsort() 辅助函数 zintersect() zunion()

辅助函数 ID ZINTERSTORE/ZUNIONSTORE

辅助函数 7-7

辅助函数 7-7

调用者可以通过传递参数来决定是否使用事务流水线。

为输入的键添加 'idx:' 前缀。

为将要被执行的操作设置好相应的参数。
如果调用者没有显式指示要延迟执行这个操作，那么实际地执行这个操作。

对有序集合执行交集计算的辅助函数。

```
def _zset_common(conn, method, scores, ttl=30, **kw):
    id = str(uuid.uuid4())
    execute = kw.pop('_execute', True)
    pipeline = conn.pipeline(True) if execute else conn
    for key in scores.keys():
        scores['idx:' + key] = scores.pop(key)

    getattr(pipeline, method)('idx:' + id, scores, **kw)
    pipeline.expire('idx:' + id, ttl)
    if execute:
        pipeline.execute()
    return id
```

```
def zintersect(conn, items, ttl=30, **kw):
    return _zset_common(conn, 'zinterstore', dict(items), ttl, **kw)

def zunion(conn, items, ttl=30, **kw):
    return _zset_common(conn, 'zunionstore', dict(items), ttl, **kw)
```

创建一个新的临时标识符。

设置事务流水线，保证每个单独的调用都有一致的结果。

为计算结果有序集合设置过期时间。

将计算结果的 ID 返回给调用者，以便做进一步的处理。

对有序集合执行并集计算的辅助函数。

7-2

1.3

1.3

article_vote() post_articles() get_articles()

get_group_articles()

composite value


```
def string_to_score(string, ignore_case=False):
    if ignore_case:
        string = string.lower()

    > pieces = map(ord, string[:6])
    while len(pieces) < 6:
        pieces.append(-1)

    score = 0
    for piece in pieces:
        score = score * 257 + piece + 1

    return score * 2 + (len(string) > 6)
```

为长度不足
位符，以此来

通过多使用一个二进制位,程序可以表明字符串是否正好为 6 个字符长,这样它就可以正确地区分出“robber”和“robbers”,尽管这对于区分“robbers”和“robbery”并无帮助。

为长度不足 6 个字符的字符串添加占位符，以此来表示这是一个短字符串。

[illegible]

7.3 数据层

数据层主要负责存储广告相关的信息，包括广告主信息、广告内容、广告位信息、广告效果数据等。数据层的设计需要考虑高并发、高可用、可扩展性等因素。

数据层主要包含以下组件：
1. ad-targeting engine：负责根据广告主的投放策略，将广告精准投放给目标用户。
2. Redis：用于缓存广告数据，提高读取效率。
3. Redis：用于缓存广告效果数据，方便实时统计。

数据层的设计需要考虑以下因素：
1. 高并发：广告系统需要处理大量的并发请求，因此数据层需要具备高并发的能力。
2. 高可用：广告系统需要保证数据的可用性，因此数据层需要具备高可用的能力。
3. 可扩展性：广告系统需要根据业务的发展进行扩展，因此数据层需要具备可扩展性的能力。
4. ad-serving platform：负责将广告内容精准投放给目标用户。
5. Web：负责接收广告主的请求，并将请求转发给数据层处理。

数据层的设计需要考虑以下因素：

7.3.1 数据层设计

数据层设计需要考虑以下因素：
1. 高并发：广告系统需要处理大量的并发请求，因此数据层需要具备高并发的能力。
2. 高可用：广告系统需要保证数据的可用性，因此数据层需要具备高可用的能力。
3. 可扩展性：广告系统需要根据业务的发展进行扩展，因此数据层需要具备可扩展性的能力。

4. CPA 和 eCPM

CPA 和 eCPM 都是 CPC 的衍生 eCPM 是衡量广告效果的一个指标，它表示每千次展示所产生的收益。CPA 是衡量广告效果的一个指标，它表示每产生一个行动所产生的成本。eCPM 和 CPA 的计算公式如下：

$$eCPM = \frac{CPA \times 1000}{1000} = CPA \times 0.2\%$$
$$CPA = \frac{CPA \times 1000}{1000} = CPA \times 0.1\%$$
$$0.002 \times 0.1 \times 3 \times 1000 = 0.60$$

图 7-9 展示了 CPC、CPA 和 eCPM 之间的关系。

图 7-9 展示了 CPC、CPA 和 eCPM 之间的关系。

```
def cpc_to_ecpm(views, clicks, cpc):  
    return 1000. * cpc * clicks / views  
  
def cpa_to_ecpm(views, actions, cpa):  
    return 1000. * cpa * actions / views
```

因为点击通过率是由点击次数除以展示次数计算出来的，而动作的执行概率则是由动作执行次数除以点击次数计算出来的，所以这两个概率相乘的结果等于动作执行次数除以展示次数。

图 7-9 展示了 CPC、CPA 和 eCPM 之间的关系。accounting system 是一个用于记录广告效果的系统。eCPM 和 CPA 都是衡量广告效果的一个指标。eCPM 是衡量广告效果的一个指标，它表示每千次展示所产生的收益。CPA 是衡量广告效果的一个指标，它表示每产生一个行动所产生的成本。eCPM 和 CPA 的计算公式如下：

图 7-9 展示了 CPC、CPA 和 eCPM 之间的关系。

7.3.3 〇〇〇〇〇〇〇〇

基于匹配的内容
计算附加值。

获取一个 ID, 它
可以用于汇报并
记录这个被定向
的广告。

记录一系列定向操作的
执行结果, 作为学习用户
行为的其中一个步骤。

```
def target_ads(conn, locations, content):  
    pipeline = conn.pipeline(True)  
    matched_ads, base_ecpm = match_location(pipeline, locations)  
    words, targeted_ads = finish_scoring(  
        pipeline, matched_ads, base_ecpm, content)
```

```
    pipeline.incr('ads:served:')  
    pipeline.zrevrange('idx:' + targeted_ads, 0, 0)  
    target_id, targeted_ad = pipeline.execute()[-2:]
```

找到 eCPM 最高的广告, 并获取这个广告 ID。

```
    if not targeted_ad:  
        return None, None
```

如果没有任何广告与目标位置
相匹配, 那么返回空值。

```
    ad_id = targeted_ad[0]  
    record_targeting_result(conn, target_id, ad_id, words)
```

```
    return target_id, ad_id
```

向调用者返回记录本次定向
操作相关信息的 ID, 以及被
选中的广告 ID。

```
def target_ads(conn, locations, content):  
    pipeline = conn.pipeline(True)  
    matched_ads, base_ecpm = match_location(pipeline, locations)  
    words, targeted_ads = finish_scoring(  
        pipeline, matched_ads, base_ecpm, content)
```

```
    pipeline.incr('ads:served:')  
    pipeline.zrevrange('idx:' + targeted_ads, 0, 0)  
    target_id, targeted_ad = pipeline.execute()[-2:]  
    if not targeted_ad:  
        return None, None  
    ad_id = targeted_ad[0]  
    record_targeting_result(conn, target_id, ad_id, words)  
    return target_id, ad_id
```

7-12 根据所有给定的位置, 找出需要执行并集操作的集合键。

根据所有给定的位置,
找出需要执行并集
操作的集合键。

找出与指定地区相匹
配的广告, 并将它们
存储到集合里面。

```
def match_location(pipe, locations):  
    required = ['req:' + loc for loc in locations]  
    matched_ads = union(pipe, required, ttl=300, _execute=False)  
    return matched_ads, zintersect(pipe,  
        {matched_ads: 0, 'ad:value:': 1}, _execute=False)
```

找到存储着所有被匹配广告的集合, 以及存
储着所有被匹配广告的基本 eCPM 的有序集
合, 然后返回它们的 ID。


```
def finish_scoring(pipe, matched, base, content):
```

```
    bonus_ecpm = {}
```

```
    words = tokenize(content)
```

```
    for word in words:
```

```
        word_bonus = zintersect(
```

```
            pipe, {matched: 0, word: 1}, _execute=False)
```

```
        bonus_ecpm[word_bonus] = 1
```

```
    if bonus_ecpm:
```

```
        minimum = zunion(
```

```
            pipe, bonus_ecpm, aggregate='MIN', _execute=False)
```

```
        maximum = zunion(
```

```
            pipe, bonus_ecpm, aggregate='MAX', _execute=False)
```

```
        return words, zunion(
```

```
            pipe, {base:1, minimum:.5, maximum:.5}, _execute=False)
```

```
    return words, base
```

→ 如果页面内容中没有出现任何可匹配的单词，那么返回广告的基本 eCPM。

对内容进行标记化处理，以便与广告进行匹配。

找出那些既位于定向位置之内，又拥有页面内容其中一个单词的广告。

计算每个广告的最小 eCPM 附加值和最大 eCPM 附加值。

将广告的基本价格、最小 eCPM 附加值的一半以及最大 eCPM 附加值的一半这三者相加起来。

```
def finish_scoring(pipe, matched, base, content):
    bonus_ecpm = {}
    words = tokenize(content)
    for word in words:
        word_bonus = zintersect(
            pipe, {matched: 0, word: 1}, _execute=False)
        bonus_ecpm[word_bonus] = 1
    if bonus_ecpm:
        minimum = zunion(
            pipe, bonus_ecpm, aggregate='MIN', _execute=False)
        maximum = zunion(
            pipe, bonus_ecpm, aggregate='MAX', _execute=False)
        return words, zunion(
            pipe, {base:1, minimum:.5, maximum:.5}, _execute=False)
    return words, base
```

7-5 7-6 7-5 7-6

Redis

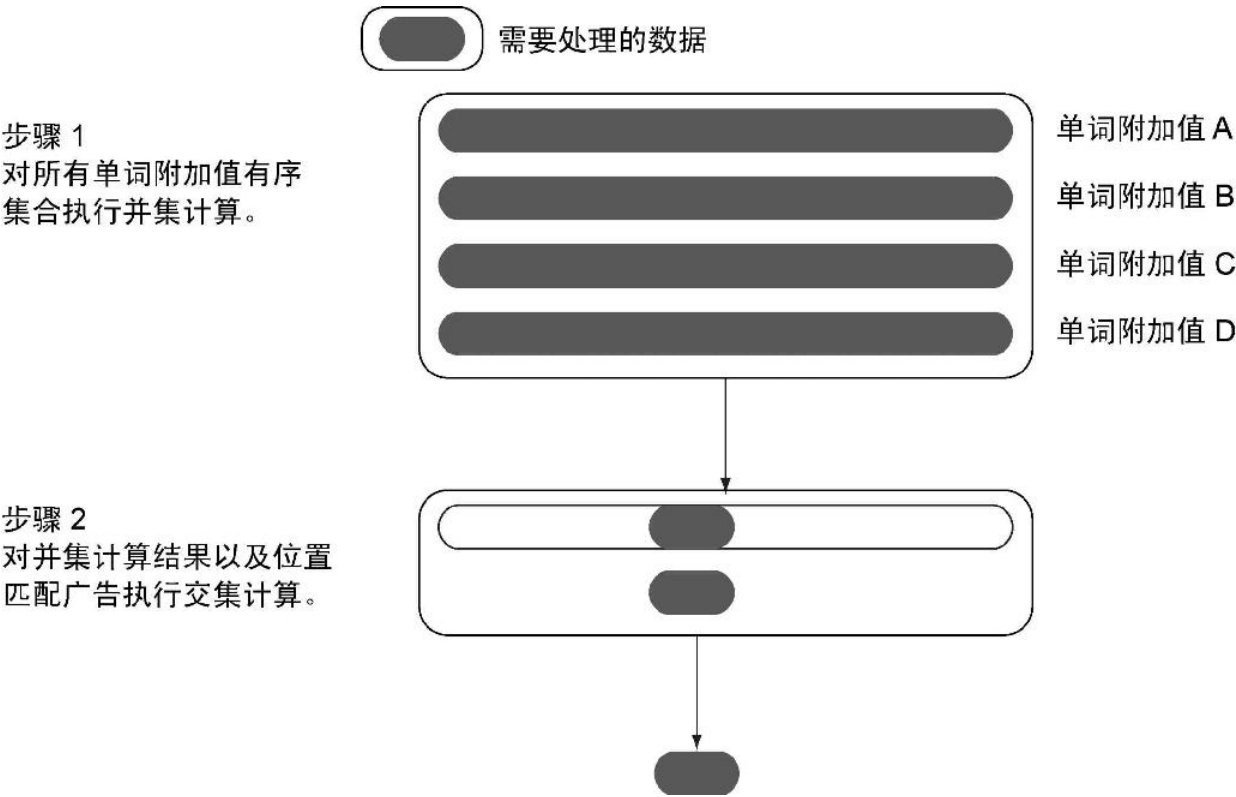


图 7-5 “位置匹配广告”与“单词附加值”的交集计算

7-11 7-13 target_ads() finish_scoring() eCPM eCPM finish_scoring()

7-11 target_ads() record_targeting_result()

7.3.4

Web Web Web

“ ” “ ”

1. 简介

本章将介绍如何记录广告展示和点击数据，并计算 eCPM。

`record_targeting_result()` 函数用于记录广告展示和点击数据。

该函数接收以下参数：
- `conn`: Redis 连接对象。
- `target_id`: 广告目标 ID。
- `ad_id`: 广告 ID。
- `words`: 广告包含的单词列表。

- 记录广告展示次数。
- 记录广告点击次数。
- 计算 eCPM。

本章将介绍如何记录广告展示和点击数据，并计算 eCPM。

本章将介绍如何记录广告展示和点击数据，并计算 eCPM。

图 7-14 记录广告展示和点击数据并计算 eCPM

```
def record_targeting_result(conn, target_id, ad_id, words):
    pipeline = conn.pipeline(True)

    terms = conn.smembers('terms:' + ad_id)
    matched = list(words & terms)
    if matched:
        matched_key = 'terms:matched:%s' % target_id
        pipeline.sadd(matched_key, *matched)
        pipeline.expire(matched_key, 900)

    type = conn.hget('type:', ad_id)
    pipeline.incr('type:%s:views:' % type)
    for word in matched:
        pipeline.zincrby('views:%s' % ad_id, word)
        pipeline.zincrby('views:%s' % ad_id, '')

    if not pipeline.execute()[-1] % 100:
        update_cpms(conn, ad_id)
```

广告每展示 100 次就更新一次它的 eCPM。

找出内容与广告之间相匹配的那些单词。

如果有相匹配的单词出现，就记录它们，并设置 15 分钟的生存时间。

为每种类型的广告分别记录它们的展示次数。

记录广告以及广告包含的单词的展示信息。

record_targeting_result() 100 update_cpms()
update_cpms() 100
100

eCPM 100
100

2

100
100
eCPM 100 × 100
100/100
0

100
100
7-15

7-15

```

def record_click(conn, target_id, ad_id, action=False):
    pipeline = conn.pipeline(True)
    click_key = 'clicks:%s'%ad_id
    match_key = 'terms:matched: %s'%target_id
    type = conn.hget('type:', ad_id)
    if type == 'cpa':
        pipeline.expire(match_key, 900)
        if action:
            click_key = 'actions:%s' % ad_id
        if action and type == 'cpa':
            pipeline.incr('type:%s:actions:' % type)
        else:
            pipeline.incr('type:%s:clicks:' % type)

    matched = list(conn.smembers(match_key))
    matched.append('')
    for word in matched:
        pipeline.zincrby(click_key, word)
    pipeline.execute()

    update_cpms(conn, ad_id)

```

根据广告的类型，维持一个全局的点击/动作计数器。

如果这是一个按动作计费的广告，并且被匹配的单词仍然存在，那么刷新这些单词的过期时间。

记录动作信息，而不是点击信息。

为广告以及所有被定向至该广告的单词记录本次点击（或动作）。

对广告中出现的所有单词的eCPM进行更新。

1. 在广告中，我们使用 `record_click()` 来记录点击/动作。
 2. 在广告中，我们使用 `update_cpms()` 来更新广告中的 eCPM。
 3. 在广告中，我们使用 `record_click()` 来记录点击/动作。

1. 在广告中，我们使用 `record_click()` 来记录点击/动作。
 2. 在广告中，我们使用 `update_cpms()` 来更新广告中的 eCPM。
 3. 在广告中，我们使用 `record_click()` 来记录点击/动作。

1. 在广告中，我们使用 `record_click()` 来记录点击/动作。
 2. 在广告中，我们使用 `update_cpms()` 来更新广告中的 eCPM。
 3. 在广告中，我们使用 `record_click()` 来记录点击/动作。

1. 在广告中，我们使用 `record_click()` 来记录点击/动作。
 2. 在广告中，我们使用 `update_cpms()` 来更新广告中的 eCPM。
 3. 在广告中，我们使用 `record_click()` 来记录点击/动作。

record_click() 7-15
1 1
finish_scoring()
record_click()

update_cpms()
[]

3 eCPM

update_cpms()
eCPM

eCPM
eCPM

eCPM
eCPM
eCPM
eCPM
eCPM

计算广告系统之程序化广告竞价策略之eCPM

图7-16 计算广告系统之程序化广告竞价策略之eCPM

图7-16 计算广告系统之程序化广告竞价策略之eCPM

```
def update_cpms(conn, ad_id):
    pipeline = conn.pipeline(True)
    pipeline.hget('type:', ad_id)
    pipeline.zscore('ad:base_value:', ad_id)
    pipeline.smembers('terms:' + ad_id)
    type, base_value, words = pipeline.execute()

    which = 'clicks'
    if type == 'cpa':
        which = 'actions'

    pipeline.get('type:%s:views:' % type)
    pipeline.get('type:%s:%s' % (type, which))
    type_views, type_clicks = pipeline.execute()
    AVERAGE_PER_1K[type] = (
        1000. * int(type_clicks or '1') / int(type_views or '1'))

    if type == 'cpm':
        return

    view_key = 'views:%s' % ad_id
    click_key = '%s:%s' % (which, ad_id)

    to_ecpm = TO_ECPM[type]

    pipeline.zscore(view_key, '')
    pipeline.zscore(click_key, '')
    ad_views, ad_clicks = pipeline.execute()
    if (ad_clicks or 0) < 1:
        ad_ecpm = conn.zscore('idx:ad:value:', ad_id)
    else:
        ad_ecpm = to_ecpm(ad_views or 1, ad_clicks or 0, base_value)
    pipeline.zadd('idx:ad:value:', ad_id, ad_ecpm)

    for word in words:
        pipeline.zscore(view_key, word)
        pipeline.zscore(click_key, word)
        views, clicks = pipeline.execute()[-2:]

        if (clicks or 0) < 1:
            continue

        word_ecpm = to_ecpm(views or 1, clicks or 0, base_value)
        bonus = word_ecpm - ad_ecpm
        pipeline.zadd('idx:' + word, ad_id, bonus)
    pipeline.execute()
```

将广告的点
击率或动作
执行率重新
写入全局字
典里面。

如果广告还没
有被点击过，
那么使用已有
的 eCPM。

计算广告
的 eCPM 并
更新它的
价格。

计算单词
的 eCPM。

获取广告的类型和价格，以及
广告包含的所有单词。

判断广告的 eCPM 应该基于点击
次数进行计算还是基于动作执行
次数进行计算。

根据给定广告的类型，获取
广告的展示次数和点击次
数（或者动作执行次数）。

如果正在处理的是一个 CPM 广告，
那么它的 eCPM 已经更新完毕，无
需再做其他处理。

获取每个广告的展示次数和点
击次数（或者动作执行次数）。

获取单词的展示次数和点击
次数（或者动作执行次数）。

如果广告还未被点击过，那么
不对 eCPM 进行更新。

计算单词的附加值。

将单词的附加值重新写
入为广告包含的每个单
词分别记录附加值的有
序集合里面。

□□□□eCPM□□

```
update_cpms() Redis
3
update_cpms()
update_cpms()
Redis 3
```

```
update_cpms()
# eCPM
# eCPM
```

Redis

- `rescale_viewed()` `global expected CTR`
- `rescale_viewed()` `global expected CTR`

7.4 ☐☐☐☐

[illegible]

Redis

7.4.1 〇〇〇〇〇〇〇〇〇〇

SDIFF

7-17

7-17

```
def add_job(conn, job_id, required_skills):
    conn.sadd('job:' + job_id, *required_skills)
def is_qualified(conn, job_id, candidate_skills):
    temp = str(uuid.uuid4())
    pipeline = conn.pipeline(True)
    pipeline.sadd(temp, *candidate_skills)
    pipeline.expire(temp, 5)
    pipeline.sdiff('job:' + job_id, temp)
    return not pipeline.execute()[-1]
```

把职位所需的技能全部添加到职位对应的集合里面。

把求职者拥有的技能全部添加到一个临时集合里面，并设置过期时间。

找出职位所需技能当中，求职者不具备的那些技能，并将它们记录到结果集合里面。

如果求职者具备职位所需的全部技能，那么返回 True。

is_qualified() 返回 True 表示求职者具备职位所需的全部技能，返回 False 表示求职者不具备职位所需的全部技能。

在 is_qualified() 函数中，我们使用了一个临时集合来存储候选人的技能，并使用 expire() 方法设置该集合的过期时间为 5 秒。这样做的目的是为了避免在检查候选人是否具备职位所需技能时，如果候选人技能集合发生变化，导致检查结果不准确。

7.4.2 索引

在 Redis 中，索引是一种特殊的数据结构，用于快速查找数据。索引通常由一个或多个键组成，每个键指向一个或多个值。索引可以用于加速对大量数据的查询操作。

在 Redis 中，索引的实现基于哈希表。每个索引项都是一个键值对，其中键是索引键，值是索引值。索引键通常是一个或多个字段的组合，索引值通常是主键或主键的某个部分。

7-18 索引结构示意图

图 7-18 索引结构示意图

```
def index_job(conn, job_id, skills):
    pipeline = conn.pipeline(True)
    for skill in skills:
        pipeline.sadd('idx:skill:' + skill, job_id)
    pipeline.zadd('idx:jobs:req', job_id, len(set(skills)))
    pipeline.execute()
```

将职位 ID 添加到相应的技能集合里面。

将职位所需技能的数量添加到记录了所有职位所需技能数量的有序集合里面。

在 7.1 节中，我们介绍了如何使用 Redis 的 pipeline 功能来批量执行命令。在 7.4.2 节中，我们将介绍如何使用 Redis 的索引功能来加速对大量数据的查询操作。

index_job() 函数用于为给定的职位 ID 创建索引。该函数接受三个参数：conn（Redis 连接对象）、job_id（职位 ID）和 skills（职位所需的技能列表）。该函数的主要逻辑如下：

redis 7.3.3

redis 7.3.3 版本中，redis 提供了 ZUNIONSTORE 命令，用于计算多个集合的并集。该命令的语法如下：

```
ZUNIONSTORE destkey numkeys key1 key2 ... keyN
```

其中，destkey 是目标集合的键名，numkeys 是参与计算的集合的个数，key1、key2、...、keyN 是参与计算的集合的键名。该命令返回的结果是目标集合的元素个数。

redis 7-19 版本中，redis 提供了 ZINTERSTORE 命令，用于计算多个集合的交集。该命令的语法如下：

```
ZINTERSTORE destkey numkeys key1 key2 ... keyN
```

redis 7-19 版本中，redis 提供了 ZINTERSTORE 命令

```
def find_jobs(conn, candidate_skills):
    skills = {}
    for skill in set(candidate_skills):
        skills['skill:' + skill] = 1
    job_scores = zunion(conn, skills)
    final_result = zintersect(
        conn, {job_scores:-1, 'jobs:req':1})
    return conn.zrangebyscore('idx:' + final_result, 0, 0)
```

计算求职者对于每个职位的得分。

设置好用于计算职位得分的字典。

计算出求职者能够胜任以及不能够胜任的职位。

返回求职者能够胜任的那些职位。

redis 7-19 版本中，redis 提供了 ZINTERSTORE 命令，用于计算多个集合的交集。该命令的语法如下：

```
ZINTERSTORE destkey numkeys key1 key2 ... keyN
```

其中，destkey 是目标集合的键名，numkeys 是参与计算的集合的个数，key1、key2、...、keyN 是参与计算的集合的键名。该命令返回的结果是目标集合的元素个数。

redis 9 版本中，redis 提供了 ZINTERSTORE 命令，用于计算多个集合的交集。该命令的语法如下：

```
ZINTERSTORE destkey numkeys key1 key2 ... keyN
```


7.5 Redis

Redis 是一个开源的、基于内存的、支持持久化的、分布式、键值对数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持主从复制、哨兵、集群等功能，使其成为一个高可用、高性能的数据库系统。

Redis 的架构非常简单，它由一个主进程和多个从进程组成。主进程负责接收客户端的请求，并将数据存储在内存中。从进程负责复制主进程的数据，并在主进程故障时接管服务。Redis 还支持多种持久化策略，如 RDB 和 AOF，以确保数据的安全。

Redis 的应用非常广泛，它被用于缓存、消息队列、分布式锁、排行榜等多种场景。例如，Twitter 使用 Redis 来存储用户的好友关系，而 Amazon 使用 Redis 来缓存商品的价格信息。

① Redis 支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持主从复制、哨兵、集群等功能，使其成为一个高可用、高性能的数据库系统。

② Redis 的架构非常简单，它由一个主进程和多个从进程组成。主进程负责接收客户端的请求，并将数据存储在内存中。从进程负责复制主进程的数据，并在主进程故障时接管服务。Redis 还支持多种持久化策略，如 RDB 和 AOF，以确保数据的安全。

③

④

ePUBw.COMePUBw.COM

8 第 8 章

本章主要介绍

- 本章主要介绍
- 本章主要介绍
- 本章主要介绍
- 本章主要介绍
- API

本章主要介绍 Twitter 的 API 接口，包括如何获取用户信息、如何获取推文、如何获取关注者信息等。本章将详细介绍 Twitter API 的各个方面，包括如何获取用户信息、如何获取推文、如何获取关注者信息等。

本章主要介绍 Twitter 的 API 接口，包括如何获取用户信息、如何获取推文、如何获取关注者信息等。本章将详细介绍 Twitter API 的各个方面，包括如何获取用户信息、如何获取推文、如何获取关注者信息等。

home timeline following list follower list

Web API streaming API

本章主要介绍 Twitter 的 API 接口，包括如何获取用户信息、如何获取推文、如何获取关注者信息等。本章将详细介绍 Twitter API 的各个方面，包括如何获取用户信息、如何获取推文、如何获取关注者信息等。

query-intensive

Twitter

[ePUBw.COM](#) [ePUBw.COM](#)

8.1 数据库

Twitter 数据库是建立在 Redis 之上的。Redis 是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Twitter 使用 Redis 来存储用户信息、推文、关注者等数据。Redis 的高性能和高可用性使得它成为 Twitter 数据库的理想选择。

Redis 数据库在 Twitter 中主要用于存储用户信息、推文、关注者等数据。Redis 的高性能和高可用性使得它成为 Twitter 数据库的理想选择。

Redis 数据库在 Twitter 中主要用于存储用户信息、推文、关注者等数据。

8.1.1 数据库

Twitter 数据库是建立在 Redis 之上的。Redis 是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Twitter 使用 Redis 来存储用户信息、推文、关注者等数据。Redis 的高性能和高可用性使得它成为 Twitter 数据库的理想选择。

Redis 数据库在 Twitter 中主要用于存储用户信息、推文、关注者等数据。Redis 的高性能和高可用性使得它成为 Twitter 数据库的理想选择。

meta-information 8-1 数据库在 Twitter 中主要用于存储用户信息、推文、关注者等数据。Redis 的高性能和高可用性使得它成为 Twitter 数据库的理想选择。

user:139960061 ————— hash	
login	dr_josiah
id	139960061
name	Josiah Carlson
followers	176
following	79
posts	386
signup	1272948506

8-1

8-1
 0
 8-1

8-1

程序使用了一个散列来存储小写的用户名以及用户 ID 之间的映射，如果给定的用户名已经被映射到了某个用户 ID，那么程序就不会再将这个用户名分配给其他人。

在散列里面将小写的用户名映射至用户 ID。

返回用户 ID。

```
def create_user(conn, login, name):
    llogin = login.lower()
    lock = acquire_lock_with_timeout(conn, 'user:' + llogin, 1)
    if not lock:
        return None

    if conn.hget('users:', llogin):
        release_lock(conn, 'user:' + llogin, lock)
        return None

    id = conn.incr('user:id:')
    pipeline = conn.pipeline(True)
    pipeline.hset('users:', llogin, id)
    pipeline.hmset('user:%s'%id, {
        'login': login,
        'id': id,
        'name': name,
        'followers': 0,
        'following': 0,
        'posts': 0,
        'signup': time.time(),
    })
    pipeline.execute()
    release_lock(conn, 'user:' + llogin, lock)
    return id
```

使用第 6 章定义的加锁函数尝试对小写的用户名进行加锁。

如果加锁不成功，那么说明给定的用户名已经被其他用户占用了。

每个用户都有一个独一无二的 ID，这个 ID 是通过对计数器执行自增操作产生的。

将用户信息添加到用户对应的散列里面。

释放之前对用户名加的锁。

```

#####
#####request#####
#####
#####ID#####ID#####
#####

```

```

#####  #####API#####
#####
#####

```

#####Twitter#####

8.1.2 数据

profile 数据

ID 数据

8-2 数据

status:223499221154799616 hash	
message	My pleasure. I was amazed that...
posted	1342908431
id	223499221154799616
uid	139960061
login	dr_josiah

8-2 数据

8-2 数据

图8-2 创建状态消息

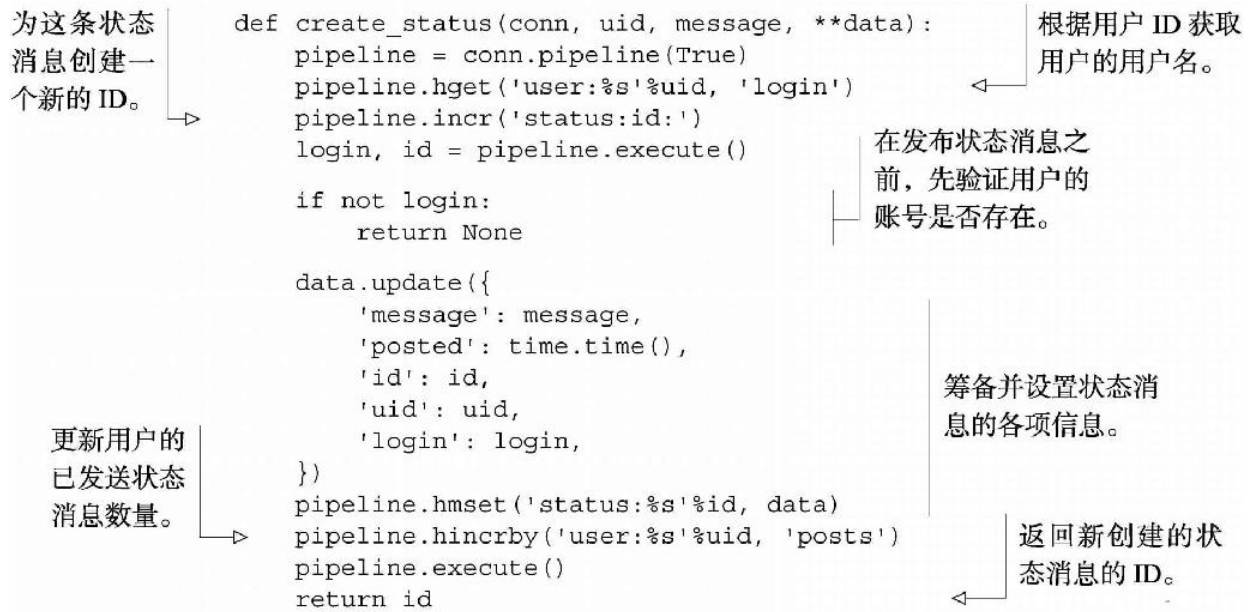


图8-2 创建状态消息

ID

图8.4

图8.4

ePUBw.COM

8.2 Redis

Redis 是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Redis 的特点包括高性能、持久化、分布式等。它广泛应用于缓存、消息队列、排行榜等场景。

Redis 的持久化机制包括 RDB 和 AOF。RDB 是快照持久化，AOF 是日志持久化。Redis 还支持主从复制、哨兵模式、集群模式等。

Redis 的集群模式可以支持高可用和水平扩展。Redis 的分布式特性使其成为构建大规模分布式系统的理想选择。

home:139960061	zset
...	...
227138379358277633	1342988984
227140001668935680	1342989371
227143088878014464	1342990107

图 8-3 Redis 集群模式下的 ID 分配

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

8.3 Redisのデータ型

Twitterのユーザーのフォロワーやフォロー中のユーザーを管理するために、Redisのデータ型として、
セット（Set）が使用されている。

ユーザーのIDを要素として持つセットを、
「followers:139960061」として管理している。

このセットには、ユーザーのIDが格納されている。IDは、
一意の識別子として使用される。図8-4は、このセットの内部構造を示している。
IDは、整数として格納されている。

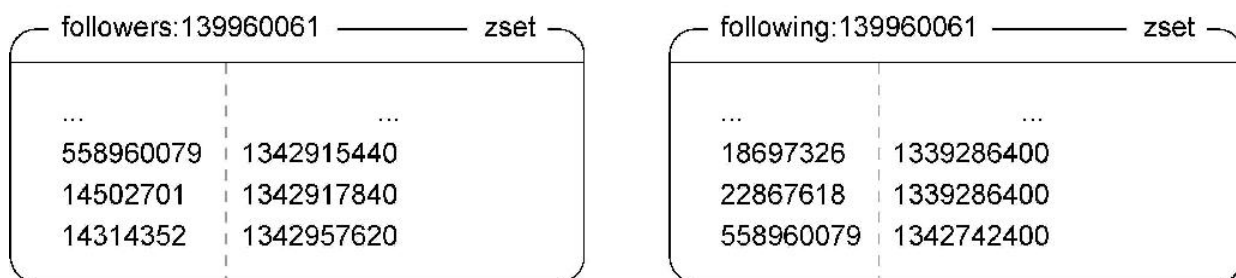


図8-4 Redisのセット（Set）の内部構造。IDは、一意の識別子として使用される。IDは、整数として格納されている。

このセットは、ユーザーのIDを要素として持つ。IDは、
一意の識別子として使用される。このセットは、ユーザーのIDを要素として持つ。

返回正在关注有序集合以及关注者有序集合的键名缓存起来。
 如果 uid 指定的用户已经关注了 other_uid 指定的用户，那么返回。
 将两个用户的 ID 分别添加到相应的正在关注有序集合以及关注者有序集合里面。
 从被关注用户的个人时间线里面获取 HOME_TIMELINE_SIZE 条最新的状态消息。
 修改两个用户的散列，更新他们各自的正在关注数量以及关注者数量。
 返回 True 表示关注操作已经成功执行。
 对执行关注操作的用户的主页时间线进行更新，并保留时间线上面的最新 1000 条状态消息。

图 8-4 实现关注操作

```

HOME_TIMELINE_SIZE = 1000
def follow_user(conn, uid, other_uid):
    fkey1 = 'following:%s'%uid
    fkey2 = 'followers:%s'%other_uid
    if conn.zscore(fkey1, other_uid):
        return None
    now = time.time()
    pipeline = conn.pipeline(True)
    pipeline.zadd(fkey1, other_uid, now)
    pipeline.zadd(fkey2, uid, now)
    pipeline.zrevrange('profile:%s'%other_uid,
        0, HOME_TIMELINE_SIZE-1, withscores=True)
    following, followers, status_and_score = pipeline.execute()[-3:]
    pipeline.hincrby('user:%s'%uid, 'following', int(following))
    pipeline.hincrby('user:%s'%other_uid, 'followers', int(followers))
    if status_and_score:
        pipeline.zadd('home:%s'%uid, **dict(status_and_score))
    pipeline.zremrangebyrank('home:%s'%uid, 0, -HOME_TIMELINE_SIZE-1)
    pipeline.execute()
    return True
    
```

实现关注操作
 返回正在关注有序集合以及关注者有序集合的键名缓存起来。
 如果 uid 指定的用户已经关注了 other_uid 指定的用户，那么返回。
 将两个用户的 ID 分别添加到相应的正在关注有序集合以及关注者有序集合里面。
 从被关注用户的个人时间线里面获取 HOME_TIMELINE_SIZE 条最新的状态消息。
 修改两个用户的散列，更新他们各自的正在关注数量以及关注者数量。
 返回 True 表示关注操作已经成功执行。
 对执行关注操作的用户的主页时间线进行更新，并保留时间线上面的最新 1000 条状态消息。

follow_user()
 返回正在关注有序集合以及关注者有序集合的键名缓存起来。
 如果 uid 指定的用户已经关注了 other_uid 指定的用户，那么返回。
 将两个用户的 ID 分别添加到相应的正在关注有序集合以及关注者有序集合里面。
 从被关注用户的个人时间线里面获取 HOME_TIMELINE_SIZE 条最新的状态消息。
 修改两个用户的散列，更新他们各自的正在关注数量以及关注者数量。
 返回 True 表示关注操作已经成功执行。
 对执行关注操作的用户的主页时间线进行更新，并保留时间线上面的最新 1000 条状态消息。

8-5

8-5

```

def unfollow_user(conn, uid, other_uid):
    fkey1 = 'following:%s'%uid
    fkey2 = 'followers:%s'%other_uid

    if not conn.zscore(fkey1, other_uid):
        return None

    pipeline = conn.pipeline(True)
    pipeline.zrem(fkey1, other_uid)
    pipeline.zrem(fkey2, uid)

    pipeline.zrevrange('profile:%s'%other_uid,
                       0, HOME_TIMELINE_SIZE-1)
    following, followers, statuses = pipeline.execute()[-3:]

    pipeline.hincrby('user:%s'%uid, 'following', int(following))
    pipeline.hincrby('user:%s'%other_uid, 'followers', int(followers))
    if statuses:
        pipeline.zrem('home:%s'%uid, *statuses)

    pipeline.execute()
    return True
    
```

如果 uid 指定的用户并未关注 other_uid 指定的用户，那么函数直接返回。

把正在关注有序集合以及关注者有序集合的键名缓存起来。

从正在关注有序集合以及关注者有序集合里面移除双方的用户 ID。

获取被取消关注的用户最近发布的 HOME_TIMELINE_SIZE 条状态消息。

对相应用户信息散列里面的正在关注数量以及关注者数量进行更新。

对执行取消关注操作的用户的主页时间线进行更新，移除被取消关注的用户发布的所有状态消息。

返回 True 表示取消关注操作执行成功。

unfollow_user()

8-5

1. 在代码中，我们使用 `Twitter` 类来创建一个新的 Twitter 对象。
 2. 然后，我们使用 `follow_user()` 方法来关注指定的用户。
 3. 最后，我们使用 `unfollow_user()` 方法来取消关注指定的用户。
 4. 在代码中，我们使用 `6.4` 来指定要关注的用户 ID。

1. 在代码中，我们使用 `Twitter` 类来创建一个新的 Twitter 对象。

1. 在代码中，我们使用 `Twitter` 类来创建一个新的 Twitter 对象。
 2. 然后，我们使用 `follow_user()` 方法来关注指定的用户。
 3. 最后，我们使用 `unfollow_user()` 方法来取消关注指定的用户。
 4. 在代码中，我们使用 `6.4` 来指定要关注的用户 ID。

1. 在代码中，我们使用 `Twitter` 类来创建一个新的 Twitter 对象。
 2. 然后，我们使用 `follow_user()` 方法来关注指定的用户。
 3. 最后，我们使用 `unfollow_user()` 方法来取消关注指定的用户。
 4. 在代码中，我们使用 `6.4` 来指定要关注的用户 ID。

1. 在代码中，我们使用 `Twitter` 类来创建一个新的 Twitter 对象。
 2. 然后，我们使用 `follow_user()` 方法来关注指定的用户。
 3. 最后，我们使用 `unfollow_user()` 方法来取消关注指定的用户。
 4. 在代码中，我们使用 `6.4` 来指定要关注的用户 ID。

1. 在代码中，我们使用 `Twitter` 类来创建一个新的 Twitter 对象。

8.4

Twitter
8.1.2
8.1.2
8.1.2

[illegible]

ID
 1000
 100 Twitter
 2500
 00

ID 1000 Twitter 1000
 10 25 10 25 0.1% 99.9%
 0.1%

1000 6.4

8-6

8-6

```
def post_status(conn, uid, message, **data):
    id = create_status(conn, uid, message, **data)
    if not id:
        return None

    posted = conn.hget('status:%s'%id, 'posted')
    if not posted:
        return None

    post = {str(id): float(posted)}
    conn.zadd('profile:%s'%uid, **post)
    syndicate_status(conn, uid, post)
    return id
```

如果创建状态消息失败，那么直接返回。

获取消息的发布时间。

将状态消息添加到用户的个人时间线里面。

使用之前介绍过的函数来创建一条新的状态消息。

如果程序未能顺利地获取消息的发布时间，那么直接返回。

将状态消息推送给用户的关注者。

post_status() 8.2

create_status()

syndicate_status()

syndicate_status() 8-7

8-7


```
POSTS_PER_PASS = 1000
def syndicate_status(conn, uid, post, start=0):
```

函数每次被调用时，最多只会将状态消息发送给 1000 个关注者。

以上次被更新的最后一个关注者为起点，获取接下来的 1000 个关注者。

将状态消息添加到所有被获取的关注者的主页时间线里面，并在有需要的时候对关注者的主页时间线进行修剪，防止它超过限定的最大长度。

```
followers = conn.zrangebyscore('followers:%s'%uid, start, 'inf',
                               start=0, num=POSTS_PER_PASS, withscores=True)
```

在遍历关注者的同时，对 `start` 变量的值进行更新，这个变量可以在有需要的时候传递给下一个 `syndicate_status()` 调用。

```
pipeline = conn.pipeline(False)
for follower, start in followers:
    pipeline.zadd('home:%s'%follower, **post)
    pipeline.zremrangebyrank(
        'home:%s'%follower, 0, -HOME_TIMELINE_SIZE-1)
pipeline.execute()
```

```
if len(followers) >= POSTS_PER_PASS:
    execute_later(conn, 'default', 'syndicate_status',
                  [conn, uid, post, start])
```

如果需要更新的关注者数量超过 1 000 人,那么在延迟任务里面继续执行剩余的更新操作。

```
syndicate_status() 1000
1000 6.4 API
post_status() syndicate_status()

```

--	--	--	--	--	--	--	--	--

[illegible]

syndicate_message()

[illegible]

`get_status_messages()`

`Python filter()`

8-8

图8-8 删除状态消息

```
def delete_status(conn, uid, status_id):
    key = 'status:%s'%status_id
    lock = acquire_lock_with_timeout(conn, key, 1)
    if not lock:
        return None

    if conn.hget(key, 'uid') != str(uid):
        release_lock(conn, key, lock)
        return None

    pipeline = conn.pipeline(True)
    pipeline.delete(key)
    pipeline.zrem('profile:%s'%uid, status_id)
    pipeline.zrem('home:%s'%uid, status_id)
    pipeline.hincrby('user:%s'%uid, 'posts', -1)
    pipeline.execute()

    release_lock(conn, key, lock)
    return True
```

对指定的状态消息进行加锁，防止两个程序同时删除同一条状态消息的情况出现。

如果加锁失败，那么直接返回。

如果uid指定的用户并非状态消息的发布人，那么函数直接返回。

从用户的个人时间线里面移除被删除状态消息的ID。

从用户的主页时间线里面移除被删除状态消息的ID。

对存储着用户信息的散列进行更新，减少已发布状态消息的数量。

删除指定的状态消息。

删除状态消息的函数 `delete_status()` 接收三个参数：连接对象 `conn`、用户ID `uid` 和状态消息ID `status_id`。函数首先尝试获取锁，如果失败则返回 `None`。然后检查状态消息是否属于该用户，如果不是则返回 `None`。接着使用管道执行删除操作，包括删除状态消息本身，并从用户的个人时间线和主页时间线中移除该消息的ID，同时更新用户的发布数量。最后释放锁并返回 `True`。

删除状态消息的ID

删除状态消息的ID是指从用户的个人时间线和主页时间线中移除该消息的ID。这可以通过使用 `zrem()` 方法来实现。在删除之前，需要先获取该消息的ID，这可以通过 `hget()` 方法从状态消息的散列中获取。

在删除状态消息之前，需要先获取该消息的ID。这可以通过使用 `hget()` 方法从状态消息的散列中获取。然后，可以使用 `zrem()` 方法从用户的个人时间线和主页时间线中移除该消息的ID。

- 删除状态消息的ID

- 訊息內容與訊息長度
- 訊息時間與時間間隔
- 訊息內容與訊息長度 conversation flow
- 訊息內容
- 訊息@訊息內容與訊息#訊息內容
- 訊息內容@訊息內容
- 訊息內容與訊息長度

訊息內容與訊息長度 Twitter 訊息內容與訊息長度
 訊息內容與訊息長度 Twitter 訊息內容與訊息長度
 訊息內容與訊息長度

- 訊息內容“”+1”
- 訊息“”訊息內容與訊息
- 訊息內容與訊息長度 6.5.2 訊息內容與訊息
- 訊息內容與訊息長度 group timeline 訊息內容與訊息
 訊息內容與訊息

訊息內容與訊息長度 Twitter 訊息內容與訊息長度
 訊息內容與訊息 API 訊息內容與訊息

訊息內容 ePUBw.COM 訊息內容 ePUBw.COM 訊息內容

訊息內容與訊息長度

8.5 API

API 是应用程序编程接口 (Application Programming Interface) 的缩写。它定义了应用程序与系统之间的交互方式。API 通常由一组函数、类或接口组成，用于实现特定的功能。API 的设计目标是提供简单、易用且可扩展的接口，以便开发人员能够快速集成和开发应用程序。

API 是应用程序编程接口 (Application Programming Interface) 的缩写。

API 是应用程序编程接口 (Application Programming Interface) 的缩写。API 是应用程序编程接口 (Application Programming Interface) 的缩写。

API 是应用程序编程接口 (Application Programming Interface) 的缩写。API 是应用程序编程接口 (Application Programming Interface) 的缩写。

API 是应用程序编程接口 (Application Programming Interface) 的缩写。

8.5.1 API 接口

API 是应用程序编程接口 (Application Programming Interface) 的缩写。API 是应用程序编程接口 (Application Programming Interface) 的缩写。

Redis Web
API
plug Web stack Web
API

API
WebSockets SPDY
chunked HTTP

Web HTTP
streamed message
data

HTTP Web Python
Python Python yield
6 Python Python
mix Web
Web Python
Python —
HTTP Web

1 HTTP

Python 是一个非常适合用于构建 HTTP 服务器的语言。它提供了许多库和框架，使得开发 HTTP 服务器变得非常简单。在本章中，我们将使用 Python 的 `socket` 库和 `threading` 库来构建一个多线程的 HTTP 服务器。我们将使用 `BaseHTTPServer` 和 `SimpleHTTPServer` 模块来创建 `request handler`，并处理 `GET` 和 `POST` 请求。本章将介绍如何构建一个 HTTP 服务器，并处理各种请求。

8-9 HTTP

让线程服务器内部组件在主服务器线程死亡之后，关闭所有客户端请求线程。

创建一个名为 `StreamingAPIServer` 的类。

这个类是一个 HTTP 服务器，并且它具有为每个请求创建一个新线程的能力。

创建一个名为 `StreamingAPIRequestHandler` 的类。

这个新创建的类可以用于处理 HTTP 请求。

调用辅助函数，获取客户端标识符。

如果这个 GET 请求访问的不是 `sample` 流或者 `firehose` 流，那么返回“404 页面未找到”错误。

如果这个 POST 请求访问的不是用户过滤器、关键字过滤器或者位置过滤器，那么返回“404 页面未找到”错误。

如果一切顺利，那么调用辅助函数，执行实际的过滤工作。

调用辅助函数，获取客户端标识符。

创建一个名为 `do_POST()` 的方法，用于处理服务器接收到的 POST 请求。

```
class StreamingAPIServer(SocketServer.ThreadingMixIn, BaseHTTPServer.HTTPServer):
    daemon_threads = True

class StreamingAPIRequestHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    def do_GET(self):
        parse_identifier(self)
        if self.path != '/statuses/sample.json':
            return self.send_error(404)
        process_filters(self)

    def do_POST(self):
        parse_identifier(self)
        if self.path != '/statuses/filter.json':
            return self.send_error(404)
        process_filters(self)
```

本章将介绍如何构建一个 HTTP 服务器，并处理各种请求。我们将使用 Python 的 `socket` 库和 `threading` 库来构建一个多线程的 HTTP 服务器。我们将使用 `BaseHTTPServer` 和 `SimpleHTTPServer` 模块来创建 `request handler`，并处理 `GET` 和 `POST` 请求。本章将介绍如何构建一个 HTTP 服务器，并处理各种请求。

图8-9 在命令行中运行API服务器

Python API 服务器在命令行中运行

API 服务器在命令行中运行

图8-10 在命令行中运行HTTP服务器

当模块在命令行里被运行的时候，运行以下代码块中的代码。

```
if __name__ == '__main__':
    server = StreamingAPIServer(
        ('localhost', 8080), StreamingAPIRequestHandler)
    print 'Starting server, use <Ctrl-C> to stop'
    server.serve_forever()
```

打印信息行。

创建一个监听本机 8080 端口的流 API 服务器实例，并使用 StreamingAPIRequestHandler 来处理请求。

服务器会一直运行直到进程被杀死为止。

API 服务器在命令行中运行

parse_identifier() 和 process_filters() 函数

API 服务器在命令行中运行

2. 在命令行中运行

图8-11 在命令行中运行 parse_identifier() 函数

在命令行中运行 parse_identifier() 函数

在命令行中运行 parse_identifier() 函数

在命令行中运行 parse_identifier() 函数

图8-11 在命令行中运行 parse_identifier() 函数

<p>如果请求里面包含了查询参数，那么处理这些参数。</p> <p>取出路径里面包含查询参数的部分，并对路径进行更新。</p> <p>获取名为 identifier 的查询参数列表。</p>	<pre>def parse_identifier(handler): handler.identifier = None handler.query = {} if '?' in handler.path: handler.path, _, query = handler.path.partition('?') handler.query = urlparse.parse_qs(query) identifier = handler.query.get('identifier') or [None] handler.identifier = identifier[0]</pre>	<p>将标识符和查询参数设置为预留值。</p> <p>通过语法分析得出查询参数。</p> <p>使用第一个传入的标识符。</p>
---	--	--

parse_identifier() 函数用于解析标识符和查询参数。它接收一个 handler 对象作为参数，并返回一个元组，其中第一个元素是标识符，第二个元素是查询参数字典。如果标识符不存在，则返回 None。

3 HTTP

HTTP 协议是万维网的数据通信的基础。它定义了客户端和服务器之间如何交换数据。HTTP 协议使用明文方式发送数据，不提供任何安全性的保障，因此可能容易被窃听、篡改和假冒。为了解决这个问题，HTTPS 协议应运而生。HTTPS 协议是在 HTTP 协议的基础上，通过 SSL/TLS 加密技术实现的。它提供了数据加密、身份认证和数据完整性保护。本章将介绍 HTTP 协议的基本概念、请求和响应格式、状态码以及 HTTPS 协议的工作原理。

8-12 本章小结

把那些需要传入参数才能使用的过滤器都放到一个列表里面。

获取客户端指定的方法，结果应该是 sample（随机消息）或者 filter（过滤器）这两种中的一种。

找到客户端在请求中指定的过滤器。

如果客户端没有指定任何过滤器，那么返回一个错误。

如果客户端指定的是随机消息请求，那么将查询参数用作 args 变量的值。

使用 Python 列表作为引用传递（pass-by-reference）变量的占位符，用户可以通过这个变量来让内容过滤器停止接收消息。

```
FILTERS = ('track', 'filter', 'location')
def process_filters(handler):
    id = handler.identifier
    if not id:
        return handler.send_error(401, "identifier missing")

    method = handler.path.rsplit('/')[-1].split('.')[0]
    name = None
    args = None
    if method == 'filter':
        data = cgi.FieldStorage(
            fp=handler.rfile,
            headers=handler.headers,
            environ={'REQUEST_METHOD': 'POST',
                     'CONTENT_TYPE': handler.headers['Content-Type'],
                     })

        for name in data:
            if name in FILTERS:
                args = data.getfirst(name).lower().split(',')
                break

        if not args:
            return handler.send_error(401, "no filter provided")
        else:
            args = handler.query

    handler.send_response(200)
    handler.send_header('Transfer-Encoding', 'chunked')
    handler.end_headers()

    quit = [False]
    for item in filter_content(id, method, name, args, quit):
        try:
            handler.wfile.write('%X\r\n%s\r\n'% (len(item), item))
        except socket.error:
            quit[0] = True
            if not quit[0]:
                handler.wfile.write('0\r\n\r\n')

    if not quit[0]:
        handler.wfile.write('0\r\n\r\n')
```

如果客户端没有提供标识符，那么返回一个错误。

如果客户端指定的是过滤器方法，那么程序需要获取相应的过滤参数。

对 POST 请求进行语法分析，从而获知过滤器的类型以及参数。

最后，向客户端返回一个回复，告知客户端，服务器接下来将向它发送流回复。

对过滤结果进行迭代。

使用分块传输编码向客户端发送经过预编码（pre-encoded）的回复。

如果服务器与客户端的连接并未断开，那么向客户端发送表示“分块到此结束”的消息。

如果发送操作引发了错误，那么让订阅者停止订阅并关闭自身。

process_filters()

HTTP

8.5.3

☐ ☐

☐ ☒ Twitter

Facebook☐ Google+☐ ☐

☐ ☐

☐ ☐ ☐ ☐ ☐ ☐

8.5.2 Web

8.5

Twitter firehose

```

        3 Redis PUBLISH SUBSCRIBE
        PUBLISH
        SUBSCRIBE
        yield back Web

```

1 □

8.1.2 8.4
8-13
create_status()

8-13 **8-2** `create_status()`

```

def create_status(conn, uid, message, **data):
    pipeline = conn.pipeline(True)
    pipeline.hget('user:%s'%uid, 'login')
    pipeline.incr('status:id:')
    login, id = pipeline.execute()

    if not login:
        return None

    data.update({
        'message': message,
        'posted': time.time(),
        'id': id,
        'uid': uid,
        'login': login,
    })
    pipeline.hmset('status:%s'%id, data)
    pipeline.hincrby('user:%s'%uid, 'posts')
    pipeline.publish('streaming:status:', json.dumps(data))
    pipeline.execute()
    return id

```

新添加的这一行代码用于向过滤器发送消息。

8-14

8-14

8-14 8-8 delete_status()

```

def delete_status(conn, uid, status_id):
    key = 'status:%s'%status_id
    lock = acquire_lock_with_timeout(conn, key, 1)

    if not lock:
        return None

    if conn.hget(key, 'uid') != str(uid):
        release_lock(conn, key, lock)
        return None

    pipeline = conn.pipeline(True)
    status = conn.hgetall(key)
    status['deleted'] = True
    pipeline.publish('streaming:status:', json.dumps(status))
    pipeline.delete(key)
    pipeline.zrem('profile:%s'%uid, status_id)
    pipeline.zrem('home:%s'%uid, status_id)
    pipeline.hincrby('user:%s'%uid, 'posts', -1)
    pipeline.execute()

    release_lock(conn, key, lock)
    return True

```

获取状态消息，以便流过滤器可以通过执行相同的过滤器来判断是否需要将被删除的消息传递给客户端。

将状态消息标记为“已被删除”。

将已被删除的状态消息发送到流里面。

```

delete_status()
"""
"""
"""
"""
ID

```

2□□□□□□□

8-15

pubsub

8-15 **8-15**

创建一个过滤器，让它来判断是否应该将消息发送给客户端。

执行订阅前的准备工作。

从订阅结构中取出状态消息。

在发送被删除的消息之前，先给消息添加一个特殊的“已被删除”占位符。

对于未被删除的匹配状态消息，程序直接发送消息本身。

```
@redis_connection('social-network')
def filter_content(conn, id, method, name, args, quit):
    match = create_filters(id, method, name, args)
```

```
    pubsub = conn.pubsub()
    pubsub.subscribe(['streaming:status:'])
```

```
    for item in pubsub.listen():
        message = item['data']
        decoded = json.loads(message)

        if match(decoded):
```

```
            if decoded.get('deleted'):
                yield json.dumps({
                    'id': decoded['id'], 'deleted': True})
```

```
            else:
                yield message
```

```
        if quit[0]:
            break
```

```
    pubsub.reset()
```

使用第 5 章中介绍的自动连接装饰器。

通过订阅来获取消息。

检查状态消息是否与过滤器相匹配。

如果 Web 服务器与客户端之间的连接已经断开，那么停止过滤消息。

重置 Redis 连接，清空因为连接速度不够快而滞留在 Redis 服务器输出缓冲区里面的数据。

filter_content()Redis
Redis
Redis

3Redisclient-output-buffer-limit
pubsub
Redis32MB
Redis

3

args 参数是一个字典，它来源于 GET 请求传递的参数。

定义一个 SampleFilter 函数，它接受 id 和 args 两个参数。

```
def SampleFilter(id, args):  
    percent = int(args.get('percent', ['10'])[0], 10)  
    ids = range(100)  
    shuffler = random.Random(id)  
    shuffler.shuffle(ids)  
    keep = set(ids[:max(percent, 1)])  
  
    def check(status):  
        return (status['id'] % 100) in keep  
    return check
```

使用 id 参数来随机地选择其中一部分消息 ID，被选中 ID 的数量由传入的 percent 参数决定。

使用 Python 集合来快速地判断给定的状态消息是否符合过滤器的标准。

创建并返回一个闭包函数，这个函数就是被创建出来的随机取样消息过滤器。

为了对状态消息进行过滤，程序会获取给定状态消息的 ID，并将 ID 的值取模 100，然后通过检查取模结果是否存在于 keep 集合来判断给定的状态消息是否符合过滤器的标准。

SampleFilter

SampleFilter 函数接受 id 参数，用于创建随机取样消息过滤器。该函数返回一个闭包函数，该闭包函数接受 status 字典作为参数，并返回布尔值。该函数使用 Python 集合来快速判断给定的状态消息是否符合过滤器的标准。该函数的时间复杂度为 $O(1)$ ，而 Python 集合的时间复杂度为 $O(n)$ 。

track 函数接受 word 参数，用于跟踪单词。该函数返回布尔值。该函数使用 8-17 和 8-18 两个函数来跟踪单词。

8-18 函数


```

def TrackFilter(list_of_strings):
    groups = []
    for group in list_of_strings:
        group = set(group.lower().split())
        if group:
            groups.append(group)
    def check(status):
        message_words = set(status['message'].lower().split())
        for group in groups:
            if len(group & message_words) == len(group):
                return True
            return False
        return check

```

以空格为分隔符，从消息里面分割出多个单词。

遍历所有词组。

函数接受一个由词组构成的列表为参数，如果一条状态消息包含某个词组里面的所有单词，那么这条消息就与过滤器相匹配。

每个词组至少需要包含一个单词。

如果某个词组的所有单词都在消息里面出现了，那么过滤器将接受这条消息。

Python Redis

Python Redis

follow

8-19 follow

8-19

```

def FollowFilter(names):
    nset = set()
    for name in names:
        nset.add('@' + name.lower().lstrip('@'))
    def check(status):
        message_words = set(status['message'].lower().split())
        message_words.add('@' + status['login'].lower())
        return message_words & nset
    return check

```

过滤器会根据给定的用户名，对消息内容以及消息的发送者进行匹配。

以“@用户名”的形式存储所有给定用户的名字。

根据消息内容以及消息发布者的名字，构建一个由空格分隔的词组。

如果给定的用户名与词组中的某个词语相同，那么这条消息与过滤器相匹配。

Python

Python

通过调用 create_status() 方法可以创建一条新的状态消息，而 post_status() 方法则是将消息发布到 Twitter 上。本章将介绍如何从 Twitter API 中获取数据，以及如何对这些数据进行处理。

8-20 位置过滤器

```
def LocationFilter(list_of_boxes):
    boxes = []
    for start in xrange(0, len(list_of_boxes)-3, 4):
        boxes.append(map(float, list_of_boxes[start:start+4]))
    def check(self, status):
        location = status.get('location')
        if not location:
            return False
        lat, lon = map(float, location.split(', '))
        for box in self.boxes:
            if (box[1] <= lat <= box[3] and
                box[0] <= lon <= box[2]):
                return True
        return False
    return check
```

创建一个区域集合，这个集合定义了过滤器接受的消息来自于哪些区域。

尝试从状态消息里面取出位置数据。

如果消息包含位置数据，那么取出纬度和经度。

如果消息未包含任何位置数据，那么这条消息不在任何区域的范围之内。

遍历所有区域，尝试进行匹配。

如果状态消息的位置在给定的经纬度范围之内，那么这条状态消息与过滤器相匹配。

本章将介绍如何从 Twitter API 中获取数据，以及如何对这些数据进行处理。本章将介绍如何从 Twitter API 中获取数据，以及如何对这些数据进行处理。

本章将介绍如何从 Twitter API 中获取数据，以及如何对这些数据进行处理。本章将介绍如何从 Twitter API 中获取数据，以及如何对这些数据进行处理。

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

8.6 架构

架构师在构建 Twitter 系统时，需要考虑如何设计一个可扩展、高性能、高可用性的系统。Twitter 系统是一个典型的分布式系统，它需要处理大量的并发请求，并且需要保证数据的实时性和一致性。在架构设计上，Twitter 系统采用了多层次的架构，包括 front end 和 back end 等组件。

在 Twitter 系统中，Redis 是一个非常重要的组件，它被用于缓存和分布式存储。Redis 是一个开源的内存数据库，它具有高性能、高可用性和易扩展性等特点。在 Twitter 系统中，Redis 被用于缓存用户信息、推文内容等数据，以提高系统的响应速度和吞吐量。

在 Twitter 系统的架构中，Redis 被用于缓存和分布式存储。Redis 是一个开源的内存数据库，它具有高性能、高可用性和易扩展性等特点。在 Twitter 系统中，Redis 被用于缓存用户信息、推文内容等数据，以提高系统的响应速度和吞吐量。此外，Redis 还被用于实现分布式锁、分布式队列等功能，以保证系统的可靠性和稳定性。

① Twitter 系统使用 API 接口接收用户请求，并将请求转发给 user stream 组件。user stream 组件负责处理用户的实时请求，并将处理结果返回给 site stream 组件。site stream 组件负责处理用户的批量请求，并将处理结果返回给 Twitter 系统的数据层。data flowing 组件负责处理系统的数据流，并将数据流转发给 Redis 数据库。Redis 数据库负责存储系统的数据，并提供高效的查询和检索功能。

<https://dev.twitter.com/streaming/public> —— 公共接口

② 在 Twitter 系统中，Redis 被用于缓存和分布式存储。Redis 是一个开源的内存数据库，它具有高性能、高可用性和易扩展性等特点。在 Twitter 系统中，Redis 被用于缓存用户信息、推文内容等数据，以提高系统的响应速度和吞吐量。

<https://dev.twitter.com/streaming/>

reference/post/statuses/filter——

③ firehose sample Twitter
sample firehose

[https://dev.twitter.com/streaming/reference/get/
statuses/sample](https://dev.twitter.com/streaming/reference/get/statuses/sample)

[https://dev.twitter.com/streaming/reference/get/statuses/fire
hose——](https://dev.twitter.com/streaming/reference/get/statuses/firehose)

ePUBw.COM ePUBw.COM

□□□□ □□□□

□□□□□□□□Redis□□□□□□□□□□□□□□□□□□□□□□□□

Lua□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□□□

9

--	--	--	--	--	--

- `struct` short structure
- `shared` structure
- `...`

① Redis 3.0 版本开始支持 Redis AOF 持久化策略。

Redis ②

3GB 1GB 70GB

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

9.1 列表

Redis 列表是 Redis 的其中一种数据类型。Redis 列表是有序的字符串集合。列表中的元素按照插入的顺序排列，你可以添加、删除列表的元素。列表的长度不受限制。列表中的元素可以是任何字符串，包括数字、布尔值、字符串等。

Redis 列表底层使用两种数据结构实现：ziplist 和 linked list。ziplist 是一种紧凑的二进制表示形式，适用于小字符串。当列表中的元素都是小字符串时，Redis 使用 ziplist。当列表中的元素包含大字符串或很多元素时，Redis 使用 linked list。linked list 是一种双向链表，每个节点包含一个指向下一个节点的指针。Redis 列表的底层实现是 skiplist，它是一种概率平衡的链表，可以快速查找元素。

9.1.1 列表操作

Redis 列表支持多种操作，包括添加、删除、获取、设置等。Redis 列表的添加操作包括 `LPUSH` 和 `RPOUSH`。Redis 列表的删除操作包括 `LPOP` 和 `RPOP`。Redis 列表的获取操作包括 `LINDEX`。Redis 列表的设置操作包括 `LSET`。Redis 列表还支持其他操作，如 `LLEN`、`LTRIM`、`LINSERT` 等。Redis 列表的底层实现是 skiplist，它是一种概率平衡的链表，可以快速查找元素。Redis 列表的底层实现是 skiplist，它是一种概率平衡的链表，可以快速查找元素。

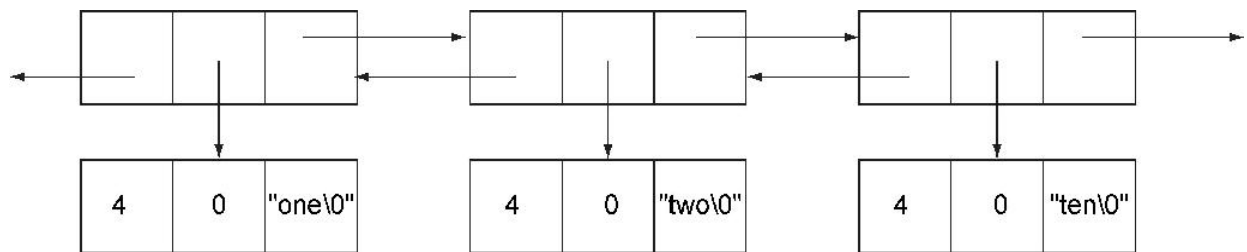


图9-1 Redis的链表结构

图9-1展示了Redis的链表结构。每个节点由三个槽位组成，中间槽位指向一个值节点。值节点包含三个槽位，分别存储数字4、0和字符串"one\0"。链表通过节点间的指针连接，形成一个双向链表结构。这种结构在Redis中用于存储字符串列表，每个字符串在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。

Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。

Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。

Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。Redis的链表结构在内存中占用一定的空间，包括字符串本身和指向下一个节点的指针。这种结构在Redis中被称为“链表”，是Redis数据结构的重要组成部分。

图9-1 列表和有序集合的限制条件

散列结构使用压缩列表表示的限制条件(Redis 2.6 以前的版本会为散列结构使用不同的编码表示,并且选项的名字也与此不同)。

```
list-max-ziplist-entries 512
list-max-ziplist-value 64
```

列表结构使用压缩列表表示的限制条件。

```
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
```

```
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
```

有序集合使用压缩列表表示的限制条件。

列表和有序集合的限制条件由以下配置项控制: `list-max-ziplist-entries`、`list-max-ziplist-value`、`hash-max-ziplist-entries`、`hash-max-ziplist-value`、`zset-max-ziplist-entries`、`zset-max-ziplist-value`。在 Redis 2.6 之前,列表和有序集合的限制条件由 `list-max-ziplist-entries`、`list-max-ziplist-value`、`hash-max-ziplist-entries`、`hash-max-ziplist-value`、`zset-max-ziplist-entries`、`zset-max-ziplist-value` 控制。

图9-1展示了 Redis 2.6 之前和 Redis 2.6 之后的限制条件。图9-2展示了 Redis 2.6 之后的限制条件。

图9-2 Redis 2.6 之后的限制条件

debug object
命令可以查看
特定对象的相
关信息。

首先将 4 个元素
推入列表。

“encoding” 信息表示这
个对象的编码为压缩列
表，这个压缩列表占用
了 24 字节内存。

再向列表中推
入 4 个元素。

对象的编码依然是
压缩列表，只是体
积增长到了 36 字节
(前面推入的 4 个元
素，每个元素都需
要花费 1 字节进行
存储，并带来 2 字
节的额外开销)。

当一个超出编码允
许大小的元素被推
入列表里面的时
候，列表将从压缩
列表编码转换为标
准的链表。

```
>>> conn.rpush('test', 'a', 'b', 'c', 'd')
4
>>> conn.debug_object('test')
{'encoding': 'ziplist', 'refcount': 1, 'lru_seconds_idle': 20,
'lru': 274841, 'at': '0xb6c9f120', 'serializedlength': 24,
'type': 'Value'}
>>> conn.rpush('test', 'e', 'f', 'g', 'h')
8
>>> conn.debug_object('test')
{'encoding': 'ziplist', 'refcount': 1, 'lru_seconds_idle': 0,
'lru': 274846, 'at': '0xb6c9f120', 'serializedlength': 36,
'type': 'Value'}
>>> conn.rpush('test', 65*'a')
9
>>> conn.debug_object('test')
{'encoding': 'linkedlist', 'refcount': 1, 'lru_seconds_idle': 10,
'lru': 274851, 'at': '0xb6c9f120', 'serializedlength': 30,
'type': 'Value'}
>>> conn.rpop('test')
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
>>> conn.debug_object('test')
{'encoding': 'linkedlist', 'refcount': 1, 'lru_seconds_idle': 0,
'lru': 274853, 'at': '0xb6c9f120', 'serializedlength': 17,
'type': 'Value'}
```

尽管序列化长度下降了，但是对于压缩列表编
码以及集合的特殊编码之外的其他编码来说，
这个数值并不代表结构的实际内存占用量。

当压缩列表被转换为普通的结构之后，即使
结构将来重新满足配置选项设置的限制条
件，结构也不会重新转换回压缩列表。

DEBUG OBJECT

9.1.2

Redis 使用整数集合表示集合

intset

Redis 使用整数集合表示集合，其内部使用 intset 结构体表示。图 9-3 展示了 intset 结构体的内部结构。

图 9-3 intset 结构体的内部结构

```
set-max-intset-entries 512
```

← 集合使用整数集合表示的限制条件。

Redis 使用整数集合表示集合，其内部使用 intset 结构体表示。图 9-4 展示了 intset 结构体的内部结构。

图 9-4 intset 结构体的内部结构

即使向集合添加 500 个元素，它的编码仍然为整数集合。

```
>>> conn.sadd('set-object', *range(500))
500
>>> conn.debug_object('set-object')
{'encoding': 'intset', 'refcount': 1, 'lru_seconds_idle': 0,
'lru': 283116, 'at': '0xb6d1a1c0', 'serializedlength': 1010,
'type': 'Value'}
>>> conn.sadd('set-object', *range(500, 1000))
500
>>> conn.debug_object('set-object')
{'encoding': 'hashtable', 'refcount': 1, 'lru_seconds_idle': 0,
'lru': 283118, 'at': '0xb6d1a1c0', 'serializedlength': 2874,
'type': 'Value'}
```

当集合的元素数量超过限定的 512 个时，整数集合将被转换为散列表表示。

9.1 Redis 使用散列表表示集合

Redis 使用散列表表示集合，其内部使用 hashtable 结构体表示。图 9-5 展示了 hashtable 结构体的内部结构。

图9-5 性能测试函数 long_ziplist_performance()

为了以不同的方式进行性能测试，函数需要对所有测试指标进行参数化处理。

删除指定的键，确保被测试数据的准确性。

通过从右端推入指定数量的元素来对列表进行初始化。

每个 rpoplpush() 函数调用都会将列表最右端的元素弹出，并将它推入同一个列表的左端。

```
def long_ziplist_performance(conn, key, length, passes, psize):
    conn.delete(key)
    conn.rpush(key, *range(length))
    pipeline = conn.pipeline(False)

    t = time.time()
    for p in xrange(passes):
        for pi in xrange(psize):
            pipeline.rpoplpush(key, key)
        pipeline.execute()

    return (passes * psize) / (time.time() - t or .001)
```

通过流水线来降低网络通信给测试带来的影响。

根据 passes 参数来决定流水线操作的执行次数。

每个流水线操作都包含了 psize 次 RPOPLPUSH 命令调用。

执行 psize 次 RPOPLPUSH 命令。

计算每秒执行的 RPOPLPUSH 调用数量。

图9-6 性能测试函数 long_ziplist_performance() 的输出结果

```
>>> long_ziplist_performance(conn, 'list', 1, 1000, 100)
52093.558416505381
>>> long_ziplist_performance(conn, 'list', 100, 1000, 100)
51501.154762768667
>>> long_ziplist_performance(conn, 'list', 1000, 1000, 100)
49732.490843316067
>>> long_ziplist_performance(conn, 'list', 5000, 1000, 100)
43424.056529592635
>>> long_ziplist_performance(conn, 'list', 10000, 1000, 100)
36727.062573334966
>>> long_ziplist_performance(conn, 'list', 50000, 1000, 100)
16695.140684975777
>>> long_ziplist_performance(conn, 'list', 100000, 500, 100)
553.10821080054586
```

当压缩列表编码的列表包含的节点数量不超过 1000 个时，Redis 每秒可以执行大约 5 万次操作。

当压缩列表编码的列表包含的节点数量达到 5000 个以上时，内存复制带来的消耗就会越来越大，导致性能下降。

当压缩列表的节点数量达到 5 万个时，性能出现明显下降。

当节点数量达到 10 万个时，压缩列表的性能低得根本没法用了。

图9-6展示了性能测试函数 long_ziplist_performance() 的输出结果。从图中可以看出，随着列表长度的增加，性能会显著下降。当列表长度达到 100000 时，性能下降得非常严重，每秒只能执行约 553 次操作。这说明了 Redis 6.1 中压缩列表的性能瓶颈。

field Redis
CPU
long_ziplist_performance() RPOPLPUSH LINDEX
LINDEX 5000
RPOPLPUSH

500 2000 128
1024
64

Redis

“”
Redis
user:joe
username:joe user username
joe
MB GB

Redis

□□□□ePUBw.COM□□□□ePUBw.COM □□□□□

□□□□□□□□□□□□

9.2 分片

分片sharding是Redis 3.0引入的一个新特性，它允许我们将一个大的数据集分成多个小的数据集，每个数据集由一个或多个节点（ shard ）来存储。分片可以显著提高Redis的吞吐量和性能，因为它可以将数据分布到多个节点上，从而避免单个节点成为瓶颈。

在Redis 3.0中，分片是通过配置参数 `redis.conf` 中的 `redis-shard` 来启用的。在9.1节中，我们介绍了Redis的集群模式，它通过X和Y来标识节点。在分片模式下，X和Y的值是shardid，它表示该节点属于哪个分片。

分片模式下，Redis的客户端需要知道每个分片的地址。Redis提供了两种方式来配置分片：一种是使用 `redis-shard` 配置文件，另一种是使用 `redis-shard` 命令。在9.1节中，我们介绍了Redis的集群模式，它通过11个节点来配置分片。在分片模式下，Redis的客户端需要知道每个分片的地址。

分片模式下，Redis的客户端需要知道每个分片的地址。Redis提供了两种方式来配置分片：一种是使用 `redis-shard` 配置文件，另一种是使用 `redis-shard` 命令。在9.1节中，我们介绍了Redis的集群模式，它通过11个节点来配置分片。在分片模式下，Redis的客户端需要知道每个分片的地址。Redis提供了两种方式来配置分片：一种是使用 `redis-shard` 配置文件，另一种是使用 `redis-shard` 命令。

分片模式下，Redis的客户端需要知道每个分片的地址。Redis提供了两种方式来配置分片：一种是使用 `redis-shard` 配置文件，另一种是使用 `redis-shard` 命令。在9.2.1节中，我们介绍了Redis的集群模式，它通过N个节点来配置分片。

redis ZADD redis ZREMRANGEBYRANK
redis

redis search index redis
redis
ZUNIONSTORE redis ZREMRANGEBYRANK redis

redis redis redis redis redis redis redis redis redis redis redis
redis redis redis redis redis redis redis redis redis redis redis
redis redis redis redis redis redis redis redis redis redis redis
redis redis redis redis redis redis redis redis redis redis redis
redis redis redis redis

redis redis redis redis redis redis redis

9.2.1 redis

redis redis redis redis redis 5.3 redis redis IP redis
redis redis redis redis IP redis redis ID redis redis redis redis
redis ID redis redis 2012 8 redis redis redis redis redis 37 redis
redis redis redis redis

redis redis redis redis redis redis redis redis redis redis redis
redis redis redis redis redis redis redis redis redis redis redis

redis集群的部署和配置

redis集群的部署和配置

resharding

9-8 shard_key() ID

HSET HGET

9-8 HSET HGET

```
def shard_hset(conn, base, key, value, total_elements, shard_size):  
    shard = shard_key(base, key, total_elements, shard_size)  
    return conn.hset(shard, key, value)
```

计算出应该由哪个分片来存储值。

将值存储到分片里面。

```
def shard_hget(conn, base, key, total_elements, shard_size):  
    shard = shard_key(base, key, total_elements, shard_size)  
    return conn.hget(shard, key)
```

计算出值可能被存储到了哪个分片里面。

取得存储在分片里面的值。

shard_hset()

shard_hget()

IP

HSET HGET shard_hset() shard_hget()

9-9

9-9 IP

为了对数据进行设置，用户需要传递TOTAL_SIZE 参数和SHARD_SIZE 参数。不过因为这个程序处理的ID都是数字，所以TOTAL_SIZE 实际上并没有被使用。

```
TOTAL_SIZE = 320000
SHARD_SIZE = 1024

def import_cities_to_redis(conn, filename):
    for row in csv.reader(open(filename)):
        ...
        shard_hset(conn, 'cityid2city:', city_id,
                    json.dumps([city, region, country]),
                    TOTAL_SIZE, SHARD_SIZE)

def find_city_by_ip(conn, ip_address):
    ...
    data = shard_hget(conn, 'cityid2city:', city_id,
                      TOTAL_SIZE, SHARD_SIZE)
    return json.loads(data)
```

把传递给分片函数的参数设置为全局常量，确保每次传递的值总是相同的。

程序在获取数据时，需要根据相同的TOTAL_SIZE 参数和SHARD_SIZE 参数查找被分片的键。

64个分片，每个分片44 MB
9-9个分片，hash-max-ziplist-entires 1024
hash-max-ziplist-value 256
150个分片，12 MB
70%的分片，3.5个分片

namespace:id
namespace:id

namespace:id

namespace:id

HDEL HINCRBY HINCRBYFLOAT

namespace:id
namespace:id

9.2.2 数据层

第1章第6章介绍了MapReduce分布式计算框架，本章将介绍MapReduce数据层，包括Redis分布式缓存、HBase分布式数据库、HDFS分布式文件系统、Hive分布式数据仓库、Pig分布式数据流处理引擎、Tez分布式任务调度引擎、Yarn分布式资源调度引擎、ZooKeeper分布式协调服务、Kafka分布式消息队列、RabbitMQ分布式消息队列、ActiveMQ分布式消息队列、RocketMQ分布式消息队列、Dubbo分布式服务框架、Spring Cloud分布式微服务框架、Kubernetes分布式容器编排引擎、Docker分布式容器引擎、Ansible分布式配置管理工具、SaltStack分布式配置管理工具、Puppet分布式配置管理工具、Chef分布式配置管理工具、Vagrant分布式虚拟化工具、Terraform分布式基础设施即代码工具、Kubernetes分布式容器编排引擎、Docker分布式容器引擎、Ansible分布式配置管理工具、SaltStack分布式配置管理工具、Puppet分布式配置管理工具、Chef分布式配置管理工具、Vagrant分布式虚拟化工具、Terraform分布式基础设施即代码工具。

本章将介绍2个分布式数据库：cookie和UUID。cookie是一个分布式数据库，用于存储用户会话信息。UUID是一个分布式数据库，用于存储用户唯一标识符。本章将介绍9.2.1节中提到的UUID数据库，并介绍其15个分布式数据库。本章将介绍15个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn，9. ZooKeeper，10. Kafka，11. RabbitMQ，12. ActiveMQ，13. RocketMQ，14. Dubbo，15. Spring Cloud。

本章将介绍UUID数据库15个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn，9. ZooKeeper，10. Kafka，11. RabbitMQ，12. ActiveMQ，13. RocketMQ，14. Dubbo，15. Spring Cloud。本章将介绍128个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn，9. ZooKeeper，10. Kafka，11. RabbitMQ，12. ActiveMQ，13. RocketMQ，14. Dubbo，15. Spring Cloud。本章将介绍16个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn，9. ZooKeeper，10. Kafka，11. RabbitMQ，12. ActiveMQ，13. RocketMQ，14. Dubbo，15. Spring Cloud。本章将介绍36个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn，9. ZooKeeper，10. Kafka，11. RabbitMQ，12. ActiveMQ，13. RocketMQ，14. Dubbo，15. Spring Cloud。本章将介绍15个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn，9. ZooKeeper，10. Kafka，11. RabbitMQ，12. ActiveMQ，13. RocketMQ，14. Dubbo，15. Spring Cloud。本章将介绍8个分布式数据库，包括：1. cookie，2. UUID，3. HBase，4. HDFS，5. Hive，6. Pig，7. Tez，8. Yarn。

——UUID 15 birthday collision 128 56 2.5 56 1% 2.5 100 1 2.5 2739 1

UUID 56 SADD 9-10 SADD 9.2.1 ID 56 ID ID 56 ID

9-10 SADD

```
def shard_sadd(conn, base, member, total_elements, shard_size):
    shard = shard_key(base,
        'x'+str(member), total_elements, shard_size)
    return conn.sadd(shard, member)
```

将成员存储到分片里面。

计算成员应该被存储到哪个分片集合里面；因为成员并非连续 ID，所以程序在计算成员所属的分片之前，会先将成员转换为字符串。

SADD UUID 56 ID ID 1 9-11

9-11

	SHARD_SIZE = 512	←	为整数集合编码的集合预设一个典型的分片大小。
取得当天的日期，并生成唯一访客计数器的键。	def count_visit(conn, session_id): today = date.today() key = 'unique:%s'%today.isoformat() expected = get_expected(conn, key, today)		
获取或者计算当天的预计唯一访客人数。	id = int(session_id.replace('-', ''[:15], 16)	←	根据 128 位的 UUID，计算出一个 56 位的 ID。
	if shard_sadd(conn, key, id, expected, SHARD_SIZE): conn.incr(key)	←	如果 ID 在分片集合里面并不存在，那么对唯一访客计数器执行加 1 操作。
将 ID 添加到分片集合里面。			

count_visit() 函数用于统计网站的访问量。

get_expected() 函数用于获取 Web 网站的预期访问量。

100 个预期访问量。

预期访问量。

预期访问量 50%。

预期访问量 2 到 9-12 个预期访问量。

预期 9-12 个预期访问量。

```

        这个初始的预计每日访客人数会
        设置得稍微比较高一些。
DAILY_EXPECTED = 1000000
EXPECTED = {}

def get_expected(conn, key, today):
    如果其他客户端已经
    计算出了当日的预计
    访客人数，那么直接使
    用已计算出的数字。
    if key in EXPECTED:
        return EXPECTED[key]
    exkey = key + ':expected'
    expected = conn.get(exkey)
    如果程序已经计算出或者获取到了当日的预
    计访客人数，那么直接使用已计算出的数字。
    获取昨天的唯一访客人数，如果该
    数值不存在就使用默认值 100 万。
    if not expected:
        yesterday = (today - timedelta(days=1)).isoformat()
        expected = conn.get('unique:%s'%yesterday)
        expected = int(expected or DAILY_EXPECTED)
    expected = 2*int(math.ceil(math.log(expected*1.5, 2)))
    基于“明天的访客人数至少会
    比今天的访客人数多 50%”
    这一假设，给昨天的访客人数
    加上 50%，然后向上舍入至
    下一个底数为 2 的幂。
    if not conn.setnx(exkey, expected):
        将计算出的预计访客人数
        写入 Redis 里面，以便其他
        程序在有需要时使用。
        expected = conn.get(exkey)
        如果在之前，已经有
        其他客户端存储了当日的
        预计访客人数，那么直接
        使用已存储的数字。
    EXPECTED[key] = int(expected)
    return EXPECTED[key]
    将当日的预计访客人数记录
    到本地副本里面，并将它返
    回给调用者。

```

get_expected() 函数实现了一个简单的算法，用于计算当日的预计访客人数。该算法基于“明天的访客人数至少会比今天的访客人数多 50%”这一假设，给昨天的访客人数加上 50%，然后向上舍入至下一个底数为 2 的幂。如果其他客户端已经计算出了当日的预计访客人数，那么直接使用已计算出的数字。如果程序已经计算出或者获取到了当日的预计访客人数，那么直接使用已计算出的数字。获取昨天的唯一访客人数，如果该数值不存在就使用默认值 100 万。如果在之前，已经有其他客户端存储了当日的预计访客人数，那么直接使用已存储的数字。将当日的预计访客人数记录到本地副本里面，并将它返回给调用者。

该算法的初始值设置为 100 万，存储在 Redis 中，占用空间为 9.5 MB。ID 的唯一性保证了每个 ID 只会被记录一次，占用空间为 56 MB。该算法的复杂度为 O(1)，即无论数据量多大，计算时间都是常数。该算法的精度为 83%，即计算结果与实际值的误差在 83% 以内。该算法的内存占用为 5.75 MB。

该算法的 API 接口如下：

```

SADD
SADD
SREM
SISMEMBER
size
ID
ID
SINTERSTORE
SUNIONSTORE
SDIFFSTORE

```

`ID` `UUID` `ID` `ID`

`ID`

`bitmap` [https://github.com/Doist/](https://github.com/Doist/bitmapist)

`bitmapist` `Python`

[illegible]

□□□□ePUBw.COM□□□□ePUBw.COM□□□□

□ □ □ □ □ □ □ □ □ □ □ □

9.3 □□□□□□□□

9.1

```
namespace{id[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]ID[0][0][0][0][0][0][0][0][0][0]
[0][0][0][0][0][0]}
```

Redis ID
Redis Twitter

```

#####Redis#####4#####
GETRANGE#####SETRANGE#####GETBIT#####SETBIT#####GETRANGE#####
#####SETRANGE#####
#####GETBIT#####SETBIT#####
#####4#####Redis#####
#####4#####
#####

```

9.3.1

12

3

4

26

2

9-13 ISO3

9-13

```
COUNTRIES = '''
ABW AFG AGO AIA ALA ALB AND ARE ARG ARM ASM ATA ATF ATG AUS AUT AZE BDI
BEL BEN BES BFA BGD BGR BHR BHS BIH BLM BLR BLZ BMU BOL BRA BRB BRN BTN
BVT BWA CAF CAN CCK CHE CHL CHN CIV CMR COD COG COK COL COM CPV CRI CUB
CUW CXR CYM CYP CZE DEU DJI DMA DNK DOM DZA ECU EGY ERI ESH ESP EST ETH
FIN FJI FLK FRA FRO FSM GAB GBR GEO GGY GHA GIB GIN GLP GMB GNB GNQ GRC
GRD GRL GTM GUF GUM GUY HKG HMD HND HRV HTI HUN IDN IMN IND IOT IRL IRN
IRQ ISL ISR ITA JAM JEY JOR JPN KAZ KEN KGZ KHM KIR KNA KOR KWT LAO LBN
LBR LBY LCA LIE LKA LSO LTU LUX LVA MAC MAF MAR MCC MDA MDG MDV MEX MHL
MKD MLI MLT MMR MNE MNG MNP MOZ MRT MSR MTQ MUS MWI MYS MYT NAM NCL NER
NFK NGU NIC NIU NLD NOR NPL NRU NZL OMN PAK PAN PCN PER PHL PLW PNG POL
PRI PRK PRT PRY PSE PYF QAT REU ROU RUS RWA SAU SDN SEN SGP SGS SHN SJM
SLB SLE SLV SMR SOM SPM SRB SSD STP SUR SVK SVN SWE SWZ SXM SYC SYR TCA
TCD TGO THA TJK TKL TKM TLS TON TTO TUN TUR TUV TWN TZA UGA UKR UMI URY
USA UZB VAT VCT VEN VGB VIR VNM VUT WLF WSM YEM ZAF ZMB ZWE'''
```

```
STATES = {
    'CAN': ''AB BC MB NB NL NS NT NU ON PE QC SK YT''.split(),
    'USA': ''AA AE AK AL AP AR AS AZ CA CO CT DC DE FL FM GA GU HI IA ID
IL IN KS KY LA MA MD ME MH MI MN MO MP MS MT NC ND NE NH NJ NM NV NY OH
OK OR PA PR PW RI SC SD TN TX UT VA VI VT WA WI WV WY''.split(),
}
```

一个由 ISO3 国家（或地区）编码组成的字符串表格，调用 split() 函数会根据空白对这个字符串进行分割，并将它转换为一个由国家（或地区）编码组成的列表。

9-13

split()

split())
2

139960061U.S.California
2
ISO3COUNTRIES
139960061
COUNTRIES"USA"
STATES["USA"]California"CA"
9-14get_code()
2

9-14

因为 Redis 里面的未初始化数据在返回时会被转换为空值，所以我们要将“未找到指定国家”时的返回值改为 0，并将第一个国家（或地区）的索引变为 1，以尝试取出国家（或地区）对应的州信息。像处理“未找到指定国家”时的情况一样，处理“未找到指定州”的情况。

```
def get_code(country, state):  
    cindex = bisect.bisect_left(COUNTRIES, country)  
    if cindex > len(COUNTRIES) or COUNTRIES[cindex] != country:  
        cindex = -1  
    cindex += 1  
    sindex = -1  
    if state and country in STATES:  
        states = STATES[country]  
        sindex = bisect.bisect_left(states, state)  
        if sindex > len(states) or states[sindex] != state:  
            sindex = -1  
    sindex += 1  
    return chr(cindex) + chr(sindex)
```

如果没有找到指定的州，那么索引为 0；如果找到了指定的州，那么索引大于 0。

寻找国家（或地区）对应的偏移量。
没有找到指定的国家（或地区）时，将其索引设置为-1。
寻找州对应的偏移量。
chr() 函数会将介于 0 至 255 之间的整数值转换为对应的 ASCII 字符。

取得用户所在位置的编码。	<pre> USERS_PER_SHARD = 2**20 def set_location(conn, user_id, country, state): code = get_code(country, state) </pre>	← 设置每个分片的大小。
查找分片 ID 以及用户在指定分片中的位置。	<pre> shard_id, position = divmod(user_id, USERS_PER_SHARD) offset = position * 2 </pre>	← 计算用户数据的偏移量。
将用户的位置信息存储到经过分片处理的位置表格里面。	<pre> pipe = conn.pipeline(False) pipe.setrange('location:%s'%shard_id, offset, code) tkey = str(uuid.uuid4()) pipe.zadd(tkey, 'max', user_id) pipe.zunionstore('location:max', [tkey, 'location:max'], aggregate='max') pipe.delete(tkey) pipe.execute() </pre>	对记录目前已知最大用户 ID 的有序集合进行更新。

set_location() 函数用于设置用户的位置信息。该函数接收四个参数：连接对象 conn、用户 ID user_id、国家 country 和州 state。首先，它调用 get_code 函数获取国家代码。然后，它使用 divmod 函数计算用户 ID 所在的分片 shard_id 和在该分片中的位置 position。接着，它使用 conn.pipeline 创建一个管道对象，并调用 setrange 方法将位置信息存储在 Redis 的 location 键中。最后，它使用 zadd 方法将用户 ID 添加到有序集合中，并使用 zunionstore 方法更新有序集合中的最大值。最后，它调用 execute 方法执行管道中的操作。

9.3.3 设置用户位置

在 Redis 中，我们可以使用 setrange 和 zadd 方法来实现用户位置的设置。setrange 方法用于设置字符串的某个位置的值，而 zadd 方法用于将元素添加到有序集合中。通过结合使用这两个方法，我们可以实现用户位置的设置和更新。

在 Redis 中，我们可以使用 readblocks 方法来实现批量读取数据。readblocks 方法接收两个参数：起始位置 start 和要读取的数据块数 count。通过调用 readblocks 方法，我们可以从 Redis 中批量读取数据，从而提高读取效率。

图9-16 聚合位置信息

获取目前已知的最大用户 ID, 并使用它来计算程序需要访问的最大分片 ID。

```
def aggregate_location(conn):  
    countries = defaultdict(int)  
    states = defaultdict(lambda: defaultdict(int))  
  
    max_id = int(conn.zscore('location:max', 'max'))  
    max_block = max_id // USERS_PER_SHARD
```

初始化两个特殊结构, 以便快速地对已存在的计数器以及缺失的计数器进行更新。

一个接一个地处理每个分片……

```
    for shard_id in xrange(max_block + 1):  
        for block in readblocks(conn, 'location:%s'%shard_id):  
            for offset in xrange(0, len(block)-1, 2):  
                code = block[offset:offset+2]  
                update_aggregates(countries, states, [code])  
  
    return countries, states
```

……读取分片中的每个块

从块里面提取出各个编码, 并根据编码查找原始的位置信息, 然后对这些位置信息进行聚合计算。

对聚合数据进行更新。

图9-16 聚合位置信息

6 Redis

aggregate_location() 9-17

ISO3

图9-17 聚合位置信息

对州计数器执行加 1 操作。

```
def update_aggregates(countries, states, codes):
    for code in codes:
        if len(code) != 2:
            continue
        country = ord(code[0]) - 1
        state = ord(code[1]) - 1
        if country < 0 or country >= len(COUNTRIES):
            continue
        country = COUNTRIES[country]
        countries[country] += 1
        if country not in STATES:
            continue
        if state < 0 or state >= STATES[country]:
            continue
        state = STATES[country][state]
        states[country][state] += 1
```

获取 ISO3 国家
(或地区) 编码。

如果程序没有找到指定的州信息，或者查找州信息时的偏移量不在合法的范围之内，那么跳过这个编码。

Twitter

9-18ID

[illegible]

9-18 ID

和之前一样，设置好基本的聚合数据。

查找用户位置信息所在分片的 ID，以及信息在分片中的偏移量。

每处理 1000 个请求，程序就会调用之前定义的辅助函数对聚合数据进行一次更新。

对遍历余下的最后一批用户进行处理。

```
def aggregate_location_list(conn, user_ids):
    pipe = conn.pipeline(False)
    countries = defaultdict(int)
    states = defaultdict(lambda: defaultdict(int))

    for i, user_id in enumerate(user_ids):
        shard_id, position = divmod(user_id, USERS_PER_SHARD)
        offset = position * 2

        pipe.substr('location:%s'%shard_id, offset, offset+1)

        if (i+1) % 1000 == 0:
            update_aggregates(countries, states, pipe.execute())

    update_aggregates(countries, states, pipe.execute())

    return countries, states
```

设置流水线，减少操作执行过程中与 Redis 的通信往返次数。

发送另一个被流水线包裹的命令，获取用户的位置信息。

返回聚合数据。

www.ePUBw.COM
www.ePUBw.COM

www.ePUBw.COM

9.4 Redis

Redis 是一个开源的分布式数据库，它支持多种数据类型，如字符串、哈希、列表、集合等。Redis 还支持持久化，可以将数据保存到磁盘上，防止数据丢失。

Redis 的安装非常简单，只需要下载 Redis 的源码包，然后编译安装即可。Redis 的配置文件位于 `redis.conf`，可以根据需要进行配置。

Redis 的客户端有很多，如 `redis-cli`、`redis-py`、`redis-jruby` 等。Redis 还支持多种语言，如 C、C++、Java、Python、Ruby 等。

① Redis 的持久化方式有两种：RDB 和 AOF。RDB 是将数据快照保存到磁盘上，AOF 是将 Redis 的写操作记录到日志中。

② Redis 的持久化配置在 `redis.conf` 文件中。RDB 的持久化配置包括 `save` 和 `stop-writes-on-bgsave-error`。AOF 的持久化配置包括 `appendonly` 和 `appendfsync`。

③ Redis 的 UUID 生成方式有两种：一种是使用 `uuid` 模块，另一种是使用 `uuid4()` 函数。Redis 的 UUID 格式为 `'4df07f45-ff2c-4057-9667-d925543e6ba3'`，其中 `'4df07f45-ff2c-4057-9667-` 是 36 个 ASCII 字符，`d925543e6ba3'` 是 36 个 ASCII 字符。

④ Redis 的内存管理方式有两种：一种是使用 `jemalloc`，另一种是使用 `malloc`。Redis 的内存管理配置在 `redis.conf` 文件中。Redis 的内存管理方式会影响 Redis 的性能和稳定性。

Python struct QQq

ePUBw.COM ePUBw.COM

第10章 Redis

本章主要介绍

- Redis简介
- Redis的安装与配置
- Redis的基本数据类型

Redis是一个开源的分布式数据库，它支持多种数据类型，如字符串、哈希、列表、集合等。Redis支持主从复制、哨兵、集群等功能，广泛应用于缓存、会话存储、排行榜等场景。

本章将详细介绍Redis的安装与配置，以及Redis的基本数据类型和常用命令。通过本章的学习，读者将能够搭建Redis环境，并对Redis的基本操作有深入的了解。

本章内容分为两部分：第一部分介绍Redis的安装与配置，第二部分介绍Redis的基本数据类型和常用命令。读者可以根据自己的需要，选择性地阅读相关内容。

本章内容来源于 ePUBw.COM 网站，更多资源请访问 ePUBw.COM。

本章内容仅供参考，不作为法律依据。

10.1 Redis

Figure 8 shows Twitter's Redis database architecture. The architecture consists of 30 Redis database nodes, each with 3 GB of memory. The nodes are organized into 10 groups, each containing 3 nodes. Each group is managed by a Redis master node, which is responsible for replicating data to the other nodes in the group. The master nodes are also responsible for managing the replication process, ensuring that all nodes in the group have the same data.

Figure 9 shows the Redis database architecture for a different application. The architecture consists of 4 Redis database nodes, each with 4 GB of memory. The nodes are organized into 2 groups, each containing 2 nodes. Each group is managed by a Redis master node, which is responsible for replicating data to the other nodes in the group. The master nodes are also responsible for managing the replication process, ensuring that all nodes in the group have the same data.

Figure 10 shows the Redis database architecture for a different application. The architecture consists of 4 Redis database nodes, each with 4 GB of memory. The nodes are organized into 2 groups, each containing 2 nodes. Each group is managed by a Redis master node, which is responsible for replicating data to the other nodes in the group. The master nodes are also responsible for managing the replication process, ensuring that all nodes in the group have the same data.

- Figure 9 shows the Redis database architecture for a different application. The architecture consists of 4 Redis database nodes, each with 4 GB of memory. The nodes are organized into 2 groups, each containing 2 nodes. Each group is managed by a Redis master node, which is responsible for replicating data to the other nodes in the group. The master nodes are also responsible for managing the replication process, ensuring that all nodes in the group have the same data.
- Figure 10 shows the Redis database architecture for a different application. The architecture consists of 4 Redis database nodes, each with 4 GB of memory. The nodes are organized into 2 groups, each containing 2 nodes. Each group is managed by a Redis master node, which is responsible for replicating data to the other nodes in the group. The master nodes are also responsible for managing the replication process, ensuring that all nodes in the group have the same data.
- Figure 11 shows the Redis database architecture for a different application. The architecture consists of 4 Redis database nodes, each with 4 GB of memory. The nodes are organized into 2 groups, each containing 2 nodes. Each group is managed by a Redis master node, which is responsible for replicating data to the other nodes in the group. The master nodes are also responsible for managing the replication process, ensuring that all nodes in the group have the same data.

在 Redis 中，我们使用 `redis-cli` 命令来操作数据库。
 例如，我们可以使用 `redis-cli` 来查看数据库中的键值对。
 在 Redis 中，我们使用 `redis-cli` 命令来操作数据库。
 例如，我们可以使用 `redis-cli` 来查看数据库中的键值对。

1. 在 Redis 10.3.1 版本中，Redis 的默认配置文件中，`requirepass` 配置项被注释掉了，这导致 Redis 没有设置密码。

```

4#####replication#####Redis#####
#####Redis#####
#####Redis#####slaveof host port#####
#####host#####port#####IP#####
#####Redis#####SLAVEOF host port#####
#####
#####SLAVEOF no one#####

```

Redis

redis 主从复制原理及配置

redis 主从复制原理及配置
redis 主从复制原理及配置

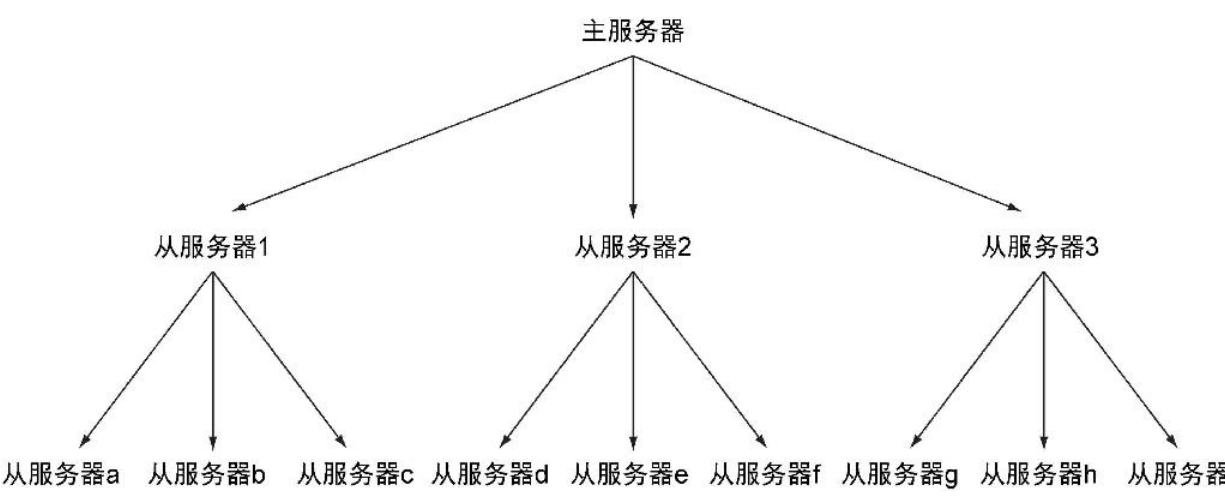


图10-1 Redis主从复制原理及配置

redis 主从复制原理及配置
redis 主从复制原理及配置
redis 主从复制原理及配置
redis 主从复制原理及配置

redis 主从复制原理及配置
redis 主从复制原理及配置
redis 主从复制原理及配置

Mbit <http://mng.bz/2ivv> SSH

配置要求：SSH、AES-128、2.6 GHz、2、AES-128、180 MB、RC4、350 MB、gigabit、SSH、gzip、SSH、1、2.6 GHz、24、52 MB、Redis、RDB、60、80 MB、Redis、AOF、5、5、5、10%、20%、1、2、3、9、1、5、10、5、1%、5%、1

OpenVPN 使用 AES 256 位元加密，OpenVPN 使用 SSH 连接，OpenVPN 使用 10 个端口，25% 30% 使用 Izo 连接。

Redis Sentinel 如何 Redis 高可用
Redis Sentinel 如何 Redis 高可用 Redis 高可用
Sentinel 如何 Redis 高可用 PUBLISH
SUBSCRIBE 如何 Redis 高可用 PING 如何 Sentinel 如何
如何 Sentinel 如何 Sentinel 如何
如何 Sentinel 如何 Sentinel 如何
如何 Sentinel 如何 Sentinel 如何
如何 Sentinel 如何 Sentinel 如何

如何 Redis Sentinel 如何
如何 Redis Sentinel 如何
如何

如何

如何 ePUBw.COM 如何 ePUBw.COM 如何

如何

10.2 Redis 部署方案

第2章介绍了 Redis 在 Web 应用中的部署方案，本章将介绍 Redis 在分布式系统中的部署方案。本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。

本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。

本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。

本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。

- 本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。
- 本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。
- 本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。
- 本章将介绍 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案，包括 Redis 在分布式系统中的部署方案。

Redis 连接池的初始化，我们使用 Redis 连接池的初始化函数。

在 9 行，我们使用 Redis 连接池的初始化函数。

在 5 行，我们使用 Redis 连接池的初始化函数。

configuration layout 使用 JSON 格式来配置 Redis。

config:redis:<component> 用于配置 Redis。

Redis 连接池的初始化函数。

10-1

10-1 Redis 连接池

```
def get_redis_connection(component, wait=1):
    key = 'config:redis:' + component
    old_config = CONFIGS.get(key, object())
    config = get_config(
        config_connection, 'redis', component, wait)
    if config != old_config:
        REDIS_CONNECTIONS[key] = redis.Redis(**config)
    return REDIS_CONNECTIONS.get(key)
```

尝试获取新的配置。

尝试获取旧的配置。

如果新旧配置不相同，那么创建一个新的连接。

返回用户指定的连接对象。

Redis 连接池的初始化函数。

Redis 连接池的初始化函数。

Redis 连接池的初始化函数。

Redis 连接池的初始化函数。

logs 7

config:redis:logs:7

10-2

10-2 分片连接池

```
def get_sharded_connection(component, key, shard_count, wait=1):  
    shard = shard_key(component, 'x'+str(key), shard_count, 2)  
    > return get_redis_connection(shard, wait)
```

返回连接。

计算出“<组件名>:<分片数字>”
格式的分片 ID。

10.2.2 分片连接池

分片连接池 Redis 分片连接池

get_sharded_connection() 分片连接池 5 分片连接池

get_sharded_connection() 分片连接池 5 分片连接池
分片连接池

5 分片连接池 3 分片连接池 5 分片连接池

分片连接池 Redis 10-3 分片连接池
分片连接池

10-3 分片连接池

创建一个包
装器，使用
它去装饰传
入的函数。

从原始函数里
面复制一些有
用的元信息到
配置处理器。

获取分片连接。

实际调用被装饰的函
数，并将分片连接以及
其他参数传递给它。

```
def sharded_connection(component, shard_count, wait=1):  
    def wrapper(function):  
        @functools.wraps(function)  
        def call(key, *args, **kwargs):  
            conn = get_sharded_connection(  
                component, key, shard_count, wait)  
            return function(conn, key, *args, **kwargs)  
        return call  
    return wrapper
```

返回一个可以对需要分片连
接的函数进行包装的函数。

装饰器接受组件名以及预期
的分片数量作为参数。

创建一个函数，
负责计算键的
分片 ID，并对
连接管理器进
行设置。

返回被包装
后的函数。

```

sharded_connection()
count_visit()count_visit()count_visit()
count_visit()
get_expected()
reuseget_expected()nonsharded
connection10-4count_visit()
get_expected()

```

```

16
JSONconfig:redis:unique:0
config:redis:unique:1516
Redisconfig:redis:unique

```

10-4 count_visit()

```

经过修改的 get_
expected()调用。
    @sharded_connection('unique', 16)
    def count_visit(conn, session_id):
        today = date.today()
        key = 'unique:%s'%today.isoformat()
        conn2, expected = get_expected(key, today)

        id = int(session_id.replace('-', '')[:15], 16)
        if shard_sadd(conn, key, id, expected, SHARD_SIZE):
            conn2.incr(key)

    @redis_connection('unique')
    def get_expected(conn, key, today):
        'all of the same function body as before, except the last line'
        return conn, EXPECTED[key]

使用 get_expect
ed() 函数返回的非
分片连接，对唯一
计数器执行自增
操作。
    将 count_visit() 函数分片
    到 16 台机器上面执行，执行所
    得的结果将被自动地分片到每
    台机器的多个数据库键上面。

    对 get_expected() 函数
    使用非分片连接。

    返回非分片连接，使得 count_visit()
    函数可以在有需要的时候，对唯一计数
    器执行自增操作。

```

```

Redis

```



```

Redis
AOF I/O Redis
AOF

```

Python <https://github.com/Doist/bitmapist>

Redis 是一个开源的、基于内存的、高性能的键值对数据库。它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Redis 还支持持久化功能，可以将内存中的数据保存到磁盘上。Redis 的架构简单，易于部署和维护。它广泛应用于缓存、会话管理、消息队列、排行榜等场景。Redis 的社区非常活跃，提供了丰富的文档和第三方库支持。

□□□□ePUBw.COM□□□□ePUBw.COM□□□□

□ □ □ □ □ □ □ □ □ □ □ □

10.3

```

Redis

```

10.3.1

[illegible]

10.1 Redis
10.1 Redis 7
SUNIONSTORE SINTERSTORE SDIFFSTORE ZINTERSTORE
ZUNIONSTORE Redis Redis 2.6
7

```

redis@redis:~$ redis-cli
redis> CONFIG GET slave-read-only
1) slave-read-only
2) yes
redis> CONFIG SET slave-read-only no
redis> CONFIG GET slave-read-only
1) slave-read-only
2) no
redis>

```

Web
Redis

10.1

scale out

10.3.2

Redis

10.3.1

index_document()

100个文档的文档ID列表，100个文档的文档ID列表，10-5个文档的文档ID列表

10-5 SORT函数

```
def search_get_values(conn, query, id=None, ttl=300, sort="-updated",
                      start=0, num=20):
```

这个函数接受的参数与 search_and_sort() 函数接受的完全相同。

```
    count, docids, id = search_and_sort(
        conn, query, id, ttl, sort, 0, start+num)

    key = "kb:doc:%s"
    sort = sort.lstrip('-')

    pipe = conn.pipeline(False)

    for docid in docids:
        pipe.hget(key%docid, sort)
    sort_column = pipe.execute()

    data_pairs = zip(docids, sort_column)
    return count, data_pairs, id
```

首先取得搜索操作和排序操作的执行结果。

根据结果的排序方式来获取数据。

将文档ID以及对文档进行排序产生的数据进行配对。

返回结果包含的文档数量、排序之后的搜索结果以及结果的缓存ID。

search_get_values() 函数返回的文档ID列表，100个文档的文档ID列表，10-5个文档的文档ID列表

search_get_values() 函数返回的文档ID列表，100个文档的文档ID列表，10-5个文档的文档ID列表

10-6

准备一些结构，
用于存储之后获
取的数据。

尝试使用已被缓存
的搜索结果；如果没
有缓存结果可用，那
么重新执行查询。

获取或者创建一个连
向指定分片的连接。

获取搜索结果以及它们
的排序数据。

```
def get_shard_results(component, shards, query, ids=None, ttl=300,
                      sort="-updated", start=0, num=20, wait=1):
    count = 0
    data = []
    ids = ids or shards * [None]
    for shard in xrange(shards):
        conn = get_redis_connection('%s:%s'%(component, shard), wait)
        c, d, i = search_get_values(
            conn, query, ids[shard], ttl, sort, start, num)

        count += c
        data.extend(d)
        ids[shard] = i
    return count, data, ids
```

程序为了获知自己要连接的服务器，
会假定所有分片服务器的信息都记
录在一个标准的配置位置里面。

将这个分片的计算结
果与所有其他分片的
计算结果进行合并。

把所有分片的原始
计算结果返回给调
用者。

get_shard_results()

get_shard_results()

Python Redis

Redis Python

Redis

get_shard_results()

missing 10-7

10-7

这个函数需要接受所有分片参数和搜索参数，这些参数大部分都会被传给底层的函数，而这个函数本身只会用到 sort 参数以及搜索偏移量。

获取未经排序的分片搜索结果。

根据 sort 参数对搜索结果进行排序。

只获取用户指定的那一页搜索结果。

```
def to_numeric_key(data):
    try:
        return Decimal(data[1] or '0')
    except:
        return Decimal('0')

def to_string_key(data):
    return data[1] or ''

def search_shards(component, shards, query, ids=None, ttl=300,
                  sort="-updated", start=0, num=20, wait=1):

    count, data, ids = get_shard_results(
        component, shards, query, ids, ttl, sort, start, num, wait)

    reversed = sort.startswith('-')
    sort = sort.strip('-')
    key = to_numeric_key
    if sort not in ('updated', 'id', 'created'):
        key = to_string_key

    data.sort(key=key, reverse=reversed)

    results = []
    for docid, score in data[start:start+num]:
        results.append(docid)

    return count, results, ids
```

这里之所以使用 Decimal 数字类型，是因为这种类型可以合理地对整数和浮点数进行转换，并在值缺失或者不是数字值的时候，返回默认值0。

总是返回一个字符中，即使在值缺失的情况下，也是如此。

准备好进行排序所需的各个参数。

返回被选中的结果，其中包括由每个分片的缓存 ID 组成的序列。

10-7 Redis Python Decimal ID

SORT Redis

2

调用 `search_and_zsort()` 函数，获取搜索结果的缓存 ID 以及结果包含的文档数量。

调用 `search_and_zsort()` 函数，获取指定的搜索结果以及这些结果的分值。

调用 `search_and_zsort()` 函数，获取指定的搜索结果以及这些结果的分值。

10-8 调用 `search_and_zsort()` 函数

```
def search_get_zset_values(conn, query, id=None, ttl=300, update=1,
                           vote=0, start=0, num=20, desc=True):
    count, r, id = search_and_zsort(
        conn, query, id, ttl, update, vote, 0, 1, desc)
    if desc:
        data = conn.zrevrange(id, 0, start + num - 1, withscores=True)
    else:
        data = conn.zrange(id, 0, start + num - 1, withscores=True)
    return count, data, id
```

这个函数接受 `search_and_zsort()` 函数所需的全部参数。

返回搜索结果的数量、搜索结果本身、搜索结果的分值以及搜索结果的缓存 ID。

调用 `search_get_zset_values()` 函数，获取指定的搜索结果以及这些结果的分值。

调用 `search_get_zset_values()` 函数，获取指定的搜索结果以及这些结果的分值。

10-9 分布式搜索函数实现

尝试使用已有的缓存结果；如果没有缓存结果可用，那么开始一次新的搜索。

获取或者创建指向每个分片的连接。

对所有搜索结果进行排序。

```
def search_shards_zset(component, shards, query, ids=None, ttl=300,
                      update=1, vote=0, start=0, num=20, desc=True, wait=1):
    count = 0
    data = []
    ids = ids or shards * [None]
    for shard in xrange(shards):
        conn = get_redis_connection('%s:%s'%(component, shard), wait)
        c, d, i = search_get_zset_values(conn, query, ids[shard],
                                         ttl, update, vote, start, num, desc)
        count += c
        data.extend(d)
        ids[shard] = i

    def key(result):
        return result[1]

    data.sort(key=key, reversed=desc)
    results = []

    for docid, score in data[start:start+num]:
        results.append(docid)

    return count, results, ids
```

准备一些结构，用于存储之后获取到的数据。

函数需要接受所有分片参数以及所有搜索参数。

对每个分片的搜索结果进行合并。

在分片上面进行搜索，并取得搜索结果的分值。

定义一个简单的排序辅助函数，让它只返回与分值有关的信息。

从结果里面提取出文档 ID，并丢弃与之关联的分值。

将搜索结果返回给调用者。

`search_shards_zset()`

分布式搜索函数实现

分布式搜索函数实现

分布式搜索函数实现

分布式搜索函数实现——Lucene Solr Elastic Search

分布式搜索函数实现Cloud Search

分布式搜索函数实现8

分布式搜索函数实现

分布式搜索函数实现

10.3.3 分布式搜索函数实现

第8章 数据库系统
数据库系统
数据库系统
数据库系统——数据库系统
数据库系统

数据库系统
数据库系统
数据库系统
数据库系统

数据库系统 数据库系统
数据库系统
数据库系统
Redis 10.3.2
数据库系统
Redis
PostgreSQL MySQL Riak MongoDB
Redis

第8章 数据库系统
数据库系统
数据库系统
数据库系统

1. 背景

在“Redis 100 个面试题”中，有一个问题是关于 Redis 的 zset-max-ziplist-size 配置项的。① 这个问题在 Redis 的官方文档中也有提到。在 Redis 2.8.19 版本之前，这个配置项的默认值是 128。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 1000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。

在 Redis 2.8.19 版本之前，这个配置项的默认值是 128。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 1000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。Twitter 在 Redis 2.8.19 版本之前，这个配置项的默认值是 1000。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 150 000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。

在 Redis 2.8.19 版本之前，这个配置项的默认值是 128。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 1000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。Twitter 在 Redis 2.8.19 版本之前，这个配置项的默认值是 1000。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 150 000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。

在 Redis 2.8.19 版本之前，这个配置项的默认值是 128。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 1000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。Twitter 在 Redis 2.8.19 版本之前，这个配置项的默认值是 1000。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 150 000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。

在 Redis 2.8.19 版本之前，这个配置项的默认值是 128。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 1000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。Twitter 在 Redis 2.8.19 版本之前，这个配置项的默认值是 1000。在 Redis 2.8.19 版本之后，这个配置项的默认值变成了 150 000。这个配置项的作用是控制 Redis 在什么情况下使用 ziplist 来存储 zset 的元素。当 zset 的元素个数小于等于这个配置项的值时，Redis 会使用 ziplist 来存储；否则，Redis 会使用 listpack 来存储。

Redis

10-10 API

10-10 API

```
sharded_timelines = KeyShardedConnection('timelines', 8)
def follow_user(conn, uid, other_uid):
    fkey1 = 'following:%s'%uid
    fkey2 = 'followers:%s'%other_uid
    if conn.zscore(fkey1, other_uid):
        print "already followed", uid, other_uid
        return None
    now = time.time()
    pipeline = conn.pipeline(True)
    pipeline.zadd(fkey1, other_uid, now)
    pipeline.zadd(fkey2, uid, now)
    pipeline.zcard(fkey1)
    pipeline.zcard(fkey2)
    following, followers = pipeline.execute()[-2:]
    pipeline.hset('user:%s'%uid, 'following', following)
    pipeline.hset('user:%s'%other_uid, 'followers', followers)
    pipeline.execute()
    pkey = 'profile:%s'%other_uid
    status_and_score = sharded_timelines[pkey].zrevrange(
        pkey, 0, HOME_TIMELINE_SIZE-1, withscores=True)
    if status_and_score:
        hkey = 'home:%s'%uid
        pipe = sharded_timelines[hkey].pipeline(True)
        pipe.zadd(hkey, **dict(status_and_score))
        pipe.zremrangebyrank(hkey, 0, -HOME_TIMELINE_SIZE-1)
        pipe.execute()
    return True
```

创建一个连接，这个连接拥有在指定分片数量的情况下，对一个组件进行分片所需的全部信息。

从正在关注的用户的个人时间线里面，取出最新的状态消息。

根据被分片的键获取一个连接，然后通过连接获取一个流水线对象。

执行事务。

将一系列状态消息添加到分片的主页时间线有序集合里面，并在添加操作完成之后对有序集合进行修剪。

API

10-11 Python

10-11

当用户尝试从对象里面
获取一个元素的时候，这
个方法就会被调用，而调
用这个方法时传入的参
数就是用户请求的元素。

```
class KeyShardedConnection(object):
    def __init__(self, component, shards):
        self.component = component
        self.shards = shards
    def __getitem__(self, key):
        return get_sharded_connection(
            self.component, key, self.shards)
```

对象使用组件名字以及
分片数量进行初始化。

根据传入的键以及之前已知的组件名字
和分片数量，获取分片连接。

Redis
unfollow_user() refill_timeline()
Redis
Redis

8
10-15
Redis

2

“
Twitter 99.99%
1000
1000


```

sharded_timelines = KeyShardedConnection('timelines', 8)
sharded_followers = KeyDataShardedConnection('followers', 16)

def follow_user(conn, uid, other_uid):
    fkey1 = 'following:%s'%uid
    fkey2 = 'followers:%s'%other_uid

    sconn = sharded_followers[uid, other_uid]
    if sconn.zscore(fkey1, other_uid):
        return None

    now = time.time()
    spipe = sconn.pipeline(True)
    spipe.zadd(fkey1, other_uid, now)
    spipe.zadd(fkey2, uid, now)
    following, followers = spipe.execute()

    pipeline = conn.pipeline(True)
    pipeline.hincrby('user:%s'%uid, 'following', int(following))
    pipeline.hincrby('user:%s'%other_uid, 'followers', int(followers))
    pipeline.execute()

    pkey = 'profile:%s'%other_uid
    status_and_score = sharded_timelines[pkey].zrevrange(
        pkey, 0, HOME_TIMELINE_SIZE-1, withscores=True)

    if status_and_score:
        hkey = 'home:%s'%uid
        pipe = sharded_timelines[hkey].pipeline(True)
        pipe.zadd(hkey, **dict(status_and_score))
        pipe.zremrangebyrank(hkey, 0, -HOME_TIMELINE_SIZE-1)
        pipe.execute()

    return True

```

创建一个连接，这个连接拥有在指定分片数量的情况下，对一个组件进行分片所需的全部信息。

根据 uid 和 other_uid 获取连接对象。

检查uid代表的用户是否已经关注了 other_uid 代表的用户。

把关注者和被关注者的信息都添加到有序集合里面。

为执行关注操作的用户以及被关注的用户更新关注者信息和正在关注信息。

10-13 使用 Redis 实现关注功能

ID 10-13 使用 Redis 实现关注功能

ID 10-13 使用 Redis 实现关注功能

10-13 ID 10-13 使用 Redis 实现关注功能

当一对 ID 作为字典查找操作的其中一个参数被传入时，这个方法将被调用。

```
class KeyDataShardedConnection(object):
    def __init__(self, component, shards):
        self.component = component
        self.shards = shards
    def __getitem__(self, ids):
        id1, id2 = map(int, ids)
        if id2 < id1:
            id1, id2 = id2, id1
        key = "%s:%s"%(id1, id2)
        return get_sharded_connection(
            self.component, key, self.shards)
```

对象使用组件名和分片数量进行初始化。

取出那对 ID，并确保它们都是整数。

基于那两个 ID 构建出一个键。

如果第二个 ID 小于第一个 ID，那么对调两个 ID 的位置，从而确保第一个 ID 总是小于等于第二个 ID。

使用构建出的键以及之前已知的组件名和分片数量，获取分片连接。

redis> hset 10-11 10 11
redis> hget 10-11 10
11
redis> hget 10-11 11
10
redis> hset 10-11 10 11
redis> hget 10-11 10
11
redis> hget 10-11 11
10

redis> zrangebyscore 10-11 10 11
10
redis> zrangebyscore 10-11 10 11
11
redis> zrangebyscore 10-11 10 11
10
redis> zrangebyscore 10-11 10 11
11
redis> zrangebyscore 10-11 10 11
10
redis> zrangebyscore 10-11 10 11
11

redis> 10.3.2
redis> 100 109
0 109
10

XXXXXXXXXXXXXXXXXXXXXXXXX10XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXX10-14XXXXXXXXXXXXXXXXXXXXZRANGEBYSCOREXXXXXXXXXX

XXXX10-14 XXXXZRANGEBYSCOREXXXXXXXXXX

获取指向当前分片的
分片连接。

函数接受组件名称、分片数量以及那些可以在分
片环境下产生正确行为的参数作为参数。

```
def sharded_zrangebyscore(component, shards, key, min, max, num):  
    data = []  
    for shard in xrange(shards):  
        conn = get_redis_connection("%s:%s"%(component, shard))  
        data.extend(conn.zrangebyscore(  
            key, min, max, start=0, num=num, withscores=True))  
  
    def key(pair):  
        return pair[1], pair[0]  
    data.sort(key=key)  
    return data[:num]
```

从 Redis 分片上
面取出数据。

首先基于分值对数据
进行排序，然后再基于
成员进行排序。

根据用户请求的
数量返回元素。

XXXXXXXXXXXX10.3.2XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX

XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXX10.3.2XX
XX
XX10-14XXXXXXXXXXXXZREVRANGEBYSCOREXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXZRANGEBYSCOREXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX10-15XXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXX

XXXX10-15 XXXXXXXXXXXXXXX

```

def syndicate_status(uid, post, start=0, on_lists=False):
    root = 'followers'
    key = 'followers:%s'%uid
    base = 'home:%s'
    if on_lists:
        root = 'list:out'
        key = 'list:out:%s'%uid
        base = 'list:statuses:%s'
    followers = sharded_zrangebyscore(root,
        sharded_followers.shards, key, start, 'inf', POSTS_PER_PASS)
    to_send = defaultdict(list)
    for follower, start in followers:
        timeline = base % follower
        shard = shard_key('timelines',
            timeline, sharded_timelines.shards, 2)
        to_send[shard].append(timeline)
    for timelines in to_send.itervalues():
        pipe = sharded_timelines[timelines[0]].pipeline(False)
        for timeline in timelines:
            pipe.zadd(timeline, **post)
            pipe.zremrangebyrank(
                timeline, 0, -HOME_TIMELINE_SIZE-1)
        pipe.execute()
    conn = redis.Redis()
    if len(followers) >= POSTS_PER_PASS:
        execute_later(conn, 'default', 'syndicate_status',
            [uid, post, start, on_lists])
    elif not on_lists:
        execute_later(conn, 'default', 'syndicate_status',
            [uid, post, 0, True])

```

基于预先分片的结果对个人信息进行分组，并把分组后的信息存储到预先准备好的结构里面。

找到负责存储这个时间线的分片。

根据存储这组时间线的服务器，找出连向它的连接，然后创建一个流水线对象。

通过 ZRANGEBYSCORE 调用，找出下一组关注者。

构造出存储时间线的键。

把时间线的键添加到位于同一个分片的其他时间线的后面。

把新发送的消息添加到时间线上面，并移除过于陈旧的消息。

1. 使用 `sharded_zrangebyscore` 方法，根据 `uid` 和 `start` 参数，从 Redis 中获取关注者的列表。

2. 使用 `defaultdict` 创建一个字典，用于存储每个关注者的时间线。

3. 遍历关注者列表，为每个关注者生成时间线键，并将其添加到字典中。

4. 遍历字典中的时间线键，为每个键创建一个 Redis 流水线对象。

5. 使用 `zadd` 方法将消息添加到时间线，并使用 `zremrangebyrank` 方法移除过期的消息。

6. 使用 `execute` 方法执行流水线。

7. 使用 `execute_later` 方法将任务添加到 Redis 的延迟队列中。

1. 使用 `sharded_zrangebyscore` 方法，根据 `uid` 和 `start` 参数，从 Redis 中获取关注者的列表。

2. 使用 `defaultdict` 创建一个字典，用于存储每个关注者的时间线。

3. 遍历关注者列表，为每个关注者生成时间线键，并将其添加到字典中。

4. 遍历字典中的时间线键，为每个键创建一个 Redis 流水线对象。

5. 使用 `zadd` 方法将消息添加到时间线，并使用 `zremrangebyrank` 方法移除过期的消息。

6. 使用 `execute` 方法执行流水线。

7. 使用 `execute_later` 方法将任务添加到 Redis 的延迟队列中。

8.4 本章小结

syndicate_status() 10-15

syndicate_status()

ePUBw.COM ePUBw.COM

10.4 Redis

Redis 是一个开源的、基于内存的、支持持久化的、分布式、键值对数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持事务、复制、集群等功能。Redis 的持久化功能可以通过 RDB 快照或 AOF 日志实现。Redis 的分布式功能可以通过 Redis Cluster 实现。Redis 的键值对数据库特性使其在缓存、消息队列、分布式锁等场景中广泛应用。

Redis 是一个开源的、基于内存的、支持持久化的、分布式、键值对数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持事务、复制、集群等功能。Redis 的持久化功能可以通过 RDB 快照或 AOF 日志实现。Redis 的分布式功能可以通过 Redis Cluster 实现。Redis 的键值对数据库特性使其在缓存、消息队列、分布式锁等场景中广泛应用。

Redis 是一个开源的、基于内存的、支持持久化的、分布式、键值对数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持事务、复制、集群等功能。Redis 的持久化功能可以通过 RDB 快照或 AOF 日志实现。Redis 的分布式功能可以通过 Redis Cluster 实现。Redis 的键值对数据库特性使其在缓存、消息队列、分布式锁等场景中广泛应用。

① Redis 是一个开源的、基于内存的、支持持久化的、分布式、键值对数据库。它支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持事务、复制、集群等功能。Redis 的持久化功能可以通过 RDB 快照或 AOF 日志实现。Redis 的分布式功能可以通过 Redis Cluster 实现。Redis 的键值对数据库特性使其在缓存、消息队列、分布式锁等场景中广泛应用。

2000 Redis 支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持事务、复制、集群等功能。Redis 的持久化功能可以通过 RDB 快照或 AOF 日志实现。Redis 的分布式功能可以通过 Redis Cluster 实现。Redis 的键值对数据库特性使其在缓存、消息队列、分布式锁等场景中广泛应用。

Redis 支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持事务、复制、集群等功能。Redis 的持久化功能可以通过 RDB 快照或 AOF 日志实现。Redis 的分布式功能可以通过 Redis Cluster 实现。Redis 的键值对数据库特性使其在缓存、消息队列、分布式锁等场景中广泛应用。

本站所有资源均来自网络，如有侵权，请联系删除。 ePUBw.COM 本站所有资源均来自网络，如有侵权，请联系删除。

本站所有资源均来自网络，如有侵权，请联系删除。

11 Redis Lua

- C
- Lua
- WATCH/MULTI/EXEC
- Lua

Redis 2.6
Lua
Redis

8
Lua
4 6
WATCH/MULTI/EXEC 6
Lua

Lua

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

11.1 □□□□C□□□□□□□□□□

Redis 2.6
Redis
10 Redis C
Redis
Redis
Redis

```

Redis Lua Lua Redis

```

[illegible]

11.1.1 Lua Redis

```

Python Redis Redis 2.6
Lua Redis
SCRIPT LOAD Lua
SHA1 EVALSHA
SHA1

```

Python Redis 脚本加载 Lua 脚本

在 Redis 中，`script_load()` 命令用于将脚本加载到 Redis 中。
 Redis 使用 `SCRIPT LOAD` 命令将脚本加载到 Redis 中。
 Redis 使用 `EVALSHA` 命令执行脚本。`11-1` 展示了 `script_load()` 的实现。

11-1 Redis 脚本加载

<p>程序只会在 SHA1 校验和未被缓存的情况下尝试载入脚本。</p>	<pre>def script_load(script): sha = [None] def call(conn, keys=[], args=[], force_eval=False): if not force_eval: if not sha[0]: sha[0] = conn.execute_command("SCRIPT", "LOAD", script, parse="LOAD") try: return conn.execute_command("EVALSHA", sha[0], len(keys), *(keys+args)) except redis.exceptions.ResponseError as msg: if not msg.args[0].startswith("NOSCRIPT"): raise return conn.execute_command("EVAL", script, len(keys), *(keys+args)) return call</pre>	<p>将 <code>SCRIPT LOAD</code> 命令返回的已载入脚本的 SHA1 校验和存储到一个列表里面，以便之后在 <code>call()</code> 函数内部对其进行修改。</p> <p>在调用已载入脚本的时候，用户需要将 Redis 连接、脚本要处理的键以及脚本的其他参数传递给脚本。</p>
<p>使用已缓存的 SHA1 校验和执行命令。</p>	<p>返回一个函数，这个函数在被调用的时候会自动载入并执行脚本。</p>	<p>当程序接收到脚本错误时，或者程序需要强制执行脚本时，它会使用 <code>EVAL</code> 命令直接执行给定的脚本。<code>EVAL</code> 命令在执行完脚本之后，会自动把脚本缓存起来，而缓存产生的 SHA1 校验和跟使用 <code>EVALSHA</code> 命令缓存脚本产生的 SHA1 校验和是完全相同的。</p>

Redis 使用 `SCRIPT LOAD` 命令将脚本加载到 Redis 中。
 Redis 使用 `EVALSHA` 命令执行脚本。`11-1` 展示了 `script_load()` 的实现。
 Redis 使用 `SCRIPT FLUSH` 命令清除 Redis 中的脚本。
 Redis 使用 `EVAL` 命令执行脚本。`EVAL` 命令在执行完脚本之后，会自动把脚本缓存起来，而缓存产生的 SHA1 校验和跟使用 `EVALSHA` 命令缓存脚本产生的 SHA1 校验和是完全相同的。

Redis 的 `script_load()` 函数可以强制 `eval` 函数
在 Redis 内部执行，而不需要连接到 Lua 解释器。

通过 **Lua** 函数，我们可以将 Redis 的 Lua 函数
集成到 Redis 3.0 版本中。Redis 的 Lua 函数
可以集成到 Redis 3.0 版本中。

Redis 的 `keys` 函数可以返回所有键名。
`keys` 函数返回的键名列表最多 10 个。
Redis 的 `keys` 函数返回的键名列表最多 10 个。

Redis 的 `script_load()` 函数可以强制 `eval` 函数
在 Redis 内部执行，而不需要连接到 Lua 解释器。

Redis 的 `script_load()` 函数可以强制 `eval` 函数
在 Redis 内部执行，而不需要连接到 Lua 解释器。

Redis 的 `script_load()` 函数可以强制 `eval` 函数
在 Redis 内部执行，而不需要连接到 Lua 解释器。

只要条件允许，
就将脚本返回
的结果转换成
相应的 Python
类型。

在大多数情况下，我们都会把脚本载入程序
返回的函数引用存储起来。

```
>>> ret_1 = script_load("return 1") <
>>> ret_1(conn)
1L
```

在此之后，我们就可以通过传入
连接对象以及脚本需要的其他
参数来调用函数。

Redis 的 `script_load()` 函数可以强制 `eval` 函数
在 Redis 内部执行，而不需要连接到 Lua 解释器。

Redis 的 `script_load()` 函数可以强制 `eval` 函数
在 Redis 内部执行，而不需要连接到 Lua 解释器。

Lua
 Lua
 11-1

Lua
 table
 Python

11-1 Lua

Lua	Python
true	1
false	None
nil	Lua
1.5	
1e30	Python
strings	
1±2 ⁵³ −1	

NO SAVE Redis Redis

AOF

8-2 create_status()

11-2

11-2 8-2

```
def create_status(conn, uid, message, **data):
    pipeline = conn.pipeline(True)
    pipeline.hget('user:%s' % uid, 'login')
    pipeline.incr('status:id:')
    login, id = pipeline.execute()

    if not login:
        return None

    data.update({
        'message': message,
        'posted': time.time(),
        'id': id,
        'uid': uid,
        'login': login,
    })
    pipeline.hmset('status:%s' % id, data)
    pipeline.hincrby('user:%s' % uid, 'posts')
    pipeline.execute()
    return id
```

根据用户 ID 获取用户的用户名。

为这条状态消息创建一个新的 ID。

在发布状态消息之前，先验证用户的账号是否存在。

更新用户的已发送状态消息数量。

准备并设置状态消息的各项信息。

返回新创建的状态消息的 ID。

Web

图 11-2 展示了 Redis 的 Lua 脚本执行流程。在 Redis 中，Lua 脚本可以通过 `eval` 命令执行。Redis 会解析脚本，并检查其是否安全。如果脚本安全，Redis 会将其交给 Lua 虚拟机执行。图 11-3 展示了 Lua 脚本在 Redis 中的执行环境。Lua 脚本在 Redis 的 Lua 虚拟机中执行，该虚拟机提供了 Redis 的 API 接口，使得 Lua 脚本可以调用 Redis 的命令。Python 也可以与 Redis 交互。

图 11-3 展示了 Redis 的 Lua 脚本执行环境。在 Redis 中，Lua 脚本是在 Lua 虚拟机中执行的。Redis 提供了 Lua 虚拟机，该虚拟机允许 Lua 脚本调用 Redis 的命令。图 11-3 展示了 Lua 脚本在 Redis 中的执行环境。Lua 脚本在 Redis 的 Lua 虚拟机中执行，该虚拟机提供了 Redis 的 API 接口，使得 Lua 脚本可以调用 Redis 的命令。Python 也可以与 Redis 交互。

图 11-3 Lua 脚本执行环境

根据用户 ID, 获取用户的用户名。记住, Lua 表格的索引是从 1 开始的, 而不是像 Python 和很多其他语言那样从 0 开始。

获取一个新的状态消息 ID。

准备好负责存储状态消息的键。

```
def create_status(conn, uid, message, **data):<
    args = [
        'message', message,
        'posted', time.time(),
        'uid', uid,
    ]
    for key, value in data.iteritems():
        args.append(key)
        args.append(value)

    return create_status_lua(
        conn, ['user:%s' % uid, 'status:id:'], args)

create_status_lua = script_load('''
local login = redis.call('hget', KEYS[1], 'login')
if not login then
    return false
end
local id = redis.call('incr', KEYS[2])
local key = string.format('status:%s', id)

redis.call('hmset', key,
    'login', login,
    'id', id,
    unpack(ARGV))
redis.call('hincrby', KEYS[1], 'posts', 1)

return id
''')
```

这个函数接受的参数和原版消息发布函数接受的参数一样。

准备好对状态消息进行设置所需的各个参数和属性。

调用脚本。

如果用户并未登录, 那么向调用者说明这一情况。

为状态消息执行数据设置操作。

对用户的已发布消息计数器执行自增操作。

返回状态消息的 ID。

在 11.1.1 节中, 我们看到了如何从 Redis 中获取 keys。在 11-3 节中, 我们看到了如何从 Redis 中获取 keys。在 11-3 节中, 我们看到了如何从 Redis 中获取 keys。在 11-3 节中, 我们看到了如何从 Redis 中获取 keys。

在 11-3 节中, 我们看到了如何从 Python 中调用 API。在 11-3 节中, 我们看到了如何从 Lua 中调用 API。在 11-3 节中, 我们看到了如何从 Lua 中调用 API。在 11-3 节中, 我们看到了如何从 Lua 中调用 API。

Redis 2.6 的 Lua 脚本支持
Python 的 redis-py 模块支持 Lua 脚本
Python Package Index
redis-py 模块支持 Lua 脚本
Redis 2.6 的 Lua 脚本支持

Redis 2.6 的 Lua 脚本支持
WATCH/MULTI/EXEC 命令支持 Lua 脚本
Redis 2.6 的 Lua 脚本支持

ePUBw.COM ePUBw.COM

Redis 2.6 的 Lua 脚本支持

11.2 Lua

6
WATCH/MULTI/EXEC
23

6.2 Lua
6.3

Lua

11.2.1 Lua

Lua

11.1.1 11.1.2 EVAL
EVALSHA Lua SHA1 Lua
Redis
WATCH/MULTI/EXEC KEYS
Redis

Redis
Redis

6.2 11-5
5 Lua

11-5 Lua acquire_lock_with_timeout()

```
def acquire_lock_with_timeout(  
    conn, lockname, acquire_timeout=10, lock_timeout=10):  
    identifier = str(uuid.uuid4())  
    lockname = 'lock:' + lockname  
    lock_timeout = int(math.ceil(lock_timeout))  
    acquired = False  
    end = time.time() + acquire_timeout  
    while time.time() < end and not acquired:  
        acquired = acquire_lock_with_timeout_lua(  
            conn, [lockname], [lock_timeout, identifier]) == 'OK'  
        time.sleep(.001 * (not acquired))  
    return acquired and identifier  
  
acquire_lock_with_timeout_lua = script_load('''  
if redis.call('exists', KEYS[1]) == 0 then  
    return redis.call('setex', KEYS[1], unpack(ARGV))  
end  
''')
```

执行实际的锁获取操作, 通过检查确保 Lua 调用已经执行成功。

检测锁是否已经存在。(再次提醒, Lua 表格的索引是从 1 开始的。)

使用给定的过期时间以及标识符去设置键。

SETNX EXPIRE SETEX
Lua
Lua

WATCH
11-6 Lua release_lock()

11-6 Lua release_lock()

```
def release_lock(conn, lockname, identifier):
    lockname = 'lock:' + lockname
    return release_lock_lua(conn, [lockname], [identifier])

release_lock_lua = script_load('''
if redis.call('get', KEYS[1]) == ARGV[1] then
    return redis.call('del', KEYS[1]) or true
end
''')
```

调用负责释放锁的 Lua 函数。

检查锁是否匹配。

删除锁并确保脚本总是返回真值。

Lua

WATCH/MULTI/EXEC

Lua

Lua

1 2 5 10

10 11-2

11-2 Lua 10

	10	10
1	31 359	31 359
2	30 085	22 507
5	47 694	19 695

11.2.3 Lua脚本

图 11-6 展示了 Redis 的 Lua 脚本。脚本首先检查信号量的持有者，如果持有者不是脚本的标识符，则脚本会尝试获取信号量。如果获取成功，脚本会返回标识符；如果获取失败，脚本会删除之前添加的标识符。

图 11-7 展示了 Redis 6.3.1 版本的 `acquire_semaphore()` 脚本。该脚本使用 Lua 脚本实现。

图 11-7 图 6.3.1 的 `acquire_semaphore()` 脚本

```
def acquire_semaphore(conn, semname, limit, timeout=10):
    identifier = str(uuid.uuid4())
    now = time.time()

    pipeline = conn.pipeline(True)
    pipeline.zremrangebyscore(semname, '-inf', now - timeout)
    pipeline.zadd(semname, identifier, now)
    pipeline.zrank(semname, identifier)
    if pipeline.execute()[-1] < limit:
        return identifier
    conn.zrem(semname, identifier)
    return None
```

清理过期的信号量持有者。

检查是否成功取得了信号量。

128 位随机标识符。

尝试获取信号量。

获取信号量失败，删除之前添加的标识符。

图 11-8 展示了 Redis 6.3.1 版本的 `acquire_semaphore()` 脚本。

6-14
 acquire_semaphore_with_lock()
 Lua
 ZINTERSTORE
 ZRANGEBYRANK

11-8 Lua acquire_semaphore()

```

def acquire_semaphore(conn, semname, limit, timeout=10):
    now = time.time()
    return acquire_semaphore_lua(conn, [semname],
        [now-timeout, limit, now, str(uuid.uuid4())])

acquire_semaphore_lua = script_load('''
redis.call('zremrangebyscore', KEYS[1], '-inf', ARGV[1])

if redis.call('zcard', KEYS[1]) < tonumber(ARGV[2]) then
    redis.call('zadd', KEYS[1], ARGV[3], ARGV[4])
    return ARGV[4]
end
''')
```

取得当前时间戳，用于处理超时信号量。

把所有必须的参数传递给 Lua 函数，实际地执行信号量获取操作。

清除所有已过期的信号量。

如果还有剩余的信号量可用，那么获取信号量。

把时间戳添加到超时有序集合里面。

6.3.1
 release_semaphore()
 6.3.3
 11-9
 Lua

11-9 Lua refresh_semaphore()

```

def refresh_semaphore(conn, semname, identifier):
    return refresh_semaphore_lua(conn, [semname],
        [identifier, time.time()]) != None

refresh_semaphore_lua = script_load('''
if redis.call('zscore', KEYS[1], ARGV[1]) then
    return redis.call('zadd', KEYS[1], ARGV[2], ARGV[1]) or true
end
''')
```

如果信号量没有被刷新，那么 Lua 脚本将返回空值，而 Python 会将这个空值转换成 None 并返回给调用者。

如果信号量仍然存在，那么对它的时间戳进行更新。

11

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

11.3 WATCH/MULTI/EXEC

WATCHMULTIEXECRedis
WATCH
Redis
WATCH
Redis
WATCH
Redis

64Lua
Redis

6

11.3.1

6

WATCH
10MULTIEXEC
11-10

11-10 6.1.2

取得范围和标识符。

```
def autocomplete_on_prefix(conn, guild, prefix):
    start, end = find_prefix_range(prefix)
    identifier = str(uuid.uuid4())

    items = autocomplete_on_prefix_lua(conn,
        ['members:' + guild],
        [start+identifier, end+identifier])

    return [item for item in items if '{' not in item]
```

使用 Lua 脚本从 Redis 里面获取数据。

过滤掉所有不想要的元素。

```
autocomplete_on_prefix_lua = script_load('''
redis.call('zadd', KEYS[1], 0, ARGV[1], 0, ARGV[2])
local sindex = redis.call('zrank', KEYS[1], ARGV[1])
local eindex = redis.call('zrank', KEYS[1], ARGV[2])
eindex = math.min(sindex + 9, eindex - 2)
```

把标记范围起点和终点的元素添加到有序集合里面。

在有序集合里面找到范围元素的位置。

```
redis.call('zrem', KEYS[1], unpack(ARGV))
return redis.call('zrange', KEYS[1], sindex, eindex)
''')
```

计算出想要获取的元素所处的范围。

移除范围元素。

获取并返回结果。

使用 Lua 脚本实现 6 个有序集合的交集。

有序集合 1 包含 26 339 个元素。

有序集合 2 包含 25 188 个元素。

有序集合 5 包含 59 544 个元素。

有序集合 11-3 包含 10 989 个元素。

图 11-3 使用 Lua 脚本实现 10 个有序集合的交集

集合	10 个集合的交集	10 个集合的并集
集合 1 包含	26 339	26 339
集合 2 包含	25 188	17 551
集合 5 包含	59 544	10 989

項目	10項目実行時	10項目実行時
10項目実行時	57 305	6 141
Lua1項目実行時	64 440	64 440
Lua2項目実行時	89 140	89 140
Lua5項目実行時	125 971	125 971
Lua10項目実行時	128 217	128 217

WATCH/MULTI/EXEC項目実行時
 10項目実行時
 Lua項目実行時
 WATCH項目実行時
 10項目実行時
 Lua20項目実行時

Lua項目実行時
 Lua項目実行時
 項目実行時

11.3.2 項目実行時

6.2 4.4 Lua

WATCH/MULTI/EXEC

Lua

11-12 6.2

11-12 6.2

```
def purchase_item_with_lock(conn, buyerid, itemid, sellerid):
    buyer = "users:%s" % buyerid
    seller = "users:%s" % sellerid
    item = "%s.%s" % (itemid, sellerid)
    inventory = "inventory:%s" % buyerid

    locked = acquire_lock(conn, 'market:')
    if not locked:
        return False

    pipe = conn.pipeline(True)
    try:
        pipe.zscore("market:", item)
        pipe.hget(buyer, 'funds')
        price, funds = pipe.execute()
        if price is None or price > funds:
            return None

        pipe.hincrby(seller, 'funds', int(price))
        pipe.hincrby(buyer, 'funds', int(-price))
        pipe.sadd(inventory, itemid)
        pipe.zrem("market:", item)
        pipe.execute()
        return True
    finally:
        release_lock(conn, 'market:', locked)
```

← 尝试获取锁。

检查商品是否已经售出，
以及买家是否有足够的
钱来购买商品。

将买家支付的钱转移
给卖家，并将售出的
商品转移给买家。

← 释放锁。

11-12

[illegible]

Lua

```
LuaWATCH/MULTI/EXEC
```

36.2.45

Lua5 Lua

11-4

11-4 **Lua** **4** **60**

	メモリサイズ	起動時間	メモリ解放時間	メモリ解放遅延
5インチ5インチディスプレイWATCH	206 000	<600	161 000	498 ms
5インチ5インチディスプレイ	21 000	20 500	0	14 ms
5インチ5インチディスプレイ	116 000	111 000	0	<3 ms
5インチ5インチディスプレイLua	505 000	480 000	0	<1 ms

```
LuaLua
```

Lua	4.25
-----	------

□□□□□□□□□□□□□□1□□□□□□□□□□□□□□0.61□□□□□□□□□□□□□□

```
00000000:  Lua                                WATCH/MULTI/EXEC
```

Lua

Redis Lua Lua Redis

WATCH/MULTI/EXEC□□□□□□□□

```
Lua
```

□ □ □ □ □

☐ ☐ ☐ ☐ **ePUBw.COM** ☐ ☐ ☐ ☐ **ePUBw.COM** ☐ ☐ ☐ ☐

[illegible]

11.4 使用 Lua 脚本

9.2 和 9.3 节中我们看到了如何编写 Lua 脚本。在 10.3 节中，我们看到了如何编写 Lua 脚本。在 11.4 节中，我们将看到如何编写 Lua 脚本。

在 9.2 节中，我们看到了如何编写 Lua 脚本。在 10.3 节中，我们看到了如何编写 Lua 脚本。在 11.4 节中，我们将看到如何编写 Lua 脚本。

在 11.4 节中，我们将看到如何编写 Lua 脚本。

11.4.1 使用 Lua 脚本

在 11.4 节中，我们将看到如何编写 Lua 脚本。在 11.4.1 节中，我们将看到如何编写 Lua 脚本。

在 11.4.1 节中，我们将看到如何编写 Lua 脚本。在 11.4.1.1 节中，我们将看到如何编写 Lua 脚本。

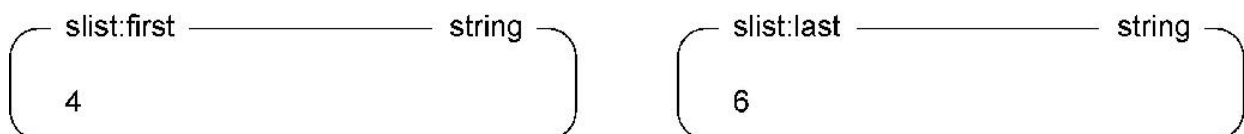


图 11-1 使用 Lua 脚本

Redis 2.6 版本开始支持 Lua 脚本，通过 Redis 2.6 的 Lua 脚本功能，可以在 Redis 服务器端执行 Lua 脚本，从而在 Redis 服务器端实现复杂的业务逻辑。

11.4.2 Redis 2.6 的 Lua 脚本

Redis 2.6 版本开始支持 Lua 脚本，通过 Redis 2.6 的 Lua 脚本功能，可以在 Redis 服务器端执行 Lua 脚本，从而在 Redis 服务器端实现复杂的业务逻辑。

Redis 2.6 版本开始支持 Lua 脚本，通过 Redis 2.6 的 Lua 脚本功能，可以在 Redis 服务器端执行 Lua 脚本，从而在 Redis 服务器端实现复杂的业务逻辑。

Redis 2.6 版本开始支持 Lua 脚本，通过 Redis 2.6 的 Lua 脚本功能，可以在 Redis 服务器端执行 Lua 脚本，从而在 Redis 服务器端实现复杂的业务逻辑。

Redis 2.6 版本开始支持 Lua 脚本，通过 Redis 2.6 的 Lua 脚本功能，可以在 Redis 服务器端执行 Lua 脚本，从而在 Redis 服务器端实现复杂的业务逻辑。

……通过调用 Lua 脚本, 把元素推入分片列表里面。

计算被推入的元素数量。

调用 sharded_push_helper() 函数, 并通过指定的参数告诉它应该执行左端推入操作还是右端推入操作。

弄清楚程序要对列表的左端还是右端进行推入, 然后取得那一端对应的分片。

取得分片的前长度。

计算出在不超过限制的情况下, 可以将多少个元素推入目前的列表里面。此外, 在列表里面保留一个节点的空间以便处理之后可能发生的阻塞弹出操作。

把元素组成的序列转换成列表。

仍然有元素需要推入时……

这个程序目前每次最多只会推入 64 个元素, 读者可以根据自己的压缩列表最大长度来调整这个数值。

移除那些已经被推入分片列表里面的元素。

返回被推入元素的总数量。

确定每个列表分片的最大长度。

如果没有元素需要进行推入, 又或者压缩列表的最大长度太小, 那么返回 0。

在条件允许的情况下, 向列表推入尽可能多的元素。

```
def sharded_push_helper(conn, key, *items, **kwargs):
    items = list(items)
    total = 0
    while items:
        pushed = sharded_push_lua(conn,
                                   [key+':', key+':first', key+':last'],
                                   [kwargs['cmd']] + items[:64])
        total += pushed
        del items[:pushed]
    return total

def sharded_lpush(conn, key, *items):
    return sharded_push_helper(conn, key, *items, cmd='lpush')

def sharded_rpush(conn, key, *items):
    return sharded_push_helper(conn, key, *items, cmd='rpush')

sharded_push_lua = script_load('''
local max = tonumber(redis.call(
    'config', 'get', 'list-max-ziplist-entries'))[2])
if #ARGV < 2 or max < 2 then return 0 end

local skey = ARGV[1] == 'lpush' and KEYS[2] or KEYS[3]
local shard = redis.call('get', skey) or '0'

while 1 do
    local current = tonumber(redis.call('llen', KEYS[1]..shard))
    local topush = math.min(#ARGV - 1, max - current - 1)
    if topush > 0 then
        redis.call(ARGV[1], KEYS[1]..shard, unpack(ARGV, 2, topush+1))
        return topush
    end
    > shard = redis.call(ARGV[1] == 'lpush' and 'decr' or 'incr', skey)
end
''')
```

否则, 生成一个新的分片并继续进行未完成的推入工作。

64

Redis

KEYS

Redis

[illegible][illegible][illegible]

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

11.4.3 □□□□□□□□

```

00000000000000000000000000000000 Lua00000000
WATCH/MULTI/ EXEC00000000000000000000000000000000
00000000WATCH/MULTI/EXEC00000000000000000000000000000000
000000000000000000000000

```

Lua

11-15

11-15 **Lua**

调整分片的端点
(endpoint)。

```
def sharded_lpop(conn, key):
    return sharded_list_pop_lua(
        conn, [key+':', key+':first', key+':last'], ['lpop'])

def sharded_rpop(conn, key):
    return sharded_list_pop_lua(
        conn, [key+':', key+':first', key+':last'], ['rpop'])

sharded_list_pop_lua = script_load('''
local skey = ARGV[1] == 'lpop' and KEYS[2] or KEYS[3]
local okey = ARGV[1] ~= 'lpop' and KEYS[2] or KEYS[3]
local shard = redis.call('get', skey) or '0'

local ret = redis.call(ARGV[1], KEYS[1]..shard)
if not ret or redis.call('llen', KEYS[1]..shard) == '0' then
    local oshard = redis.call('get', okey) or '0'

    if shard == oshard then
        return ret
    end

    local cmd = ARGV[1] == 'lpop' and 'incr' or 'decr'

    shard = redis.call(cmd, skey)
    if not ret then
        ret = redis.call(ARGV[1], KEYS[1]..shard)
    end
end
return ret
''')
```

获取需要执行弹出操作的片片的ID。

如果程序因为分片为空而没有得到弹出元素，又或者弹出操作使得分片变空了，那么对分片端点进行清理。

根据被弹出的元素来自列表的左端还是右端，决定应该增加还是减少分片的ID。

如果之前没有取得弹出元素，那么尝试对新分片进行弹出。

[illegible]

□□□□□□□□□□API□□□□□□□□□□

11.4.4 □□□□□□□□□□

[illegible]


```

Lua
Lua

```

```

    ""MULTI"/"EXEC""      MULTI/EXEC
WATCH/MULTI/EXEC
BLPOPBRPOPEXECMULTI/EXEC
BLPOPBRPOP
MULTI/EXECBLPOP
BRPOLPOP

```

[illegible]

□□□□**11-16** □□□□□□□□□□□□□□□□

预先定义好的伪元素，读者也可以按自己的需要，把这个伪元素替换成某个不可能出现在分片列表里面的值。

定义一个辅助函数，这个函数会为左端阻塞弹出操作以及右端阻塞弹出操作执行实际的弹出动作。

```
DUMMY = str(uuid.uuid4())
```

```
def sharded_bpop_helper(conn, key, timeout, pop, bpop, endp, push):
```

取得程序认为需要对其
执行弹出操作的分片。

```
pipe = conn.pipeline(False)
timeout = max(timeout, 0) or 2**64
end = time.time() + timeout
```

准备好流水线对象和超时信息。

运行 Lua 脚本辅助程序，它会在程序尝试从错误的分片里面弹出元素的时候，将一个伪元素推入那个分片里面。

```
while time.time() < end:
    result = pop(conn, key)
    if result not in (None, DUMMY):
        return result
```

尝试执行一次非阻塞弹出，如果这个操作成功取得了一个弹出值，并且这个值并不是伪元素，那么返回这个值。

```
shard = conn.get(key + endp) or '0'
sharded_bpop_helper_lua(pipe, [key + ':', key + endp],
    [shard, push, DUMMY], force_eval=True)
getattr(pipe, bpop)(key + ':' + shard, 1)
```

因为程序不能在流水线里面执行一个可能会失败的 EVALSHA 调用，所以这里需要使用 `force_eval` 参数，确保程序调用的是 EVAL 命令而不是 EVALSHA 命令。

```
result = (pipe.execute() [-1] or [None]) [-1]
if result not in (None, DUMMY):
    return result
```

如果命令返回了一个元素，那么程序执行完毕；否则的话，进行重试。

使用用户传入的 BLPOP 命令或 BRPOP 命令,对列表执行阻塞弹出操作。

这些函数负责调用底层的阻塞弹出操作。

```
def sharded_bpop(conn, key, timeout=0):
    return sharded_bpop_helper(
        conn, key, timeout, sharded_lpop, 'lpop', ':first', 'lpush')
```

如果程序接下来要从错误的分片里面弹出元素,那么将伪元素推入那个分片里面。

```
def sharded_brpop(conn, key, timeout=0):
    return sharded_bpop_helper(
        conn, key, timeout, sharded_rpop, 'brpop', ':last', 'rpush')
```

```
sharded_bpop_helper_lua = script_load(''''
local shard = redis.call('get', KEYS[2]) or '0'
if shard ~= ARGV[1] then
    redis.call(ARGV[2], KEYS[1]..ARGV[1], ARGV[3])
end
''')
```

找到程序想要对其执行弹出操作的列表端,并取得这个列表端对应的分片。

3

[illegible]

Lua

API

```
00000000000000000000WATCH/MULTI/EXEC0000000000000000
```

[illegible]

□ □ □ □ □ □ □ □

[illegible]

11.5 脚本

Redis 2.6 版本开始支持 Lua 脚本，Lua 脚本可以在 Redis 中执行，Redis 提供了 WATCH/MULTI/EXEC 命令，用于在 Lua 脚本中执行 Redis 命令。

Redis 2.6 版本开始支持 Lua 脚本，Redis 提供了 WATCH/MULTI/EXEC 命令，用于在 Lua 脚本中执行 Redis 命令。

① Redis 2.6 版本开始支持 Lua 脚本，Redis 提供了 WATCH/MULTI/EXEC 命令，用于在 Lua 脚本中执行 Redis 命令。Redis 2.6 版本开始支持 Lua 脚本，Redis 提供了 WATCH/MULTI/EXEC 命令，用于在 Lua 脚本中执行 Redis 命令。

本站所有资源均来自网络，如有侵权，请联系删除。
ePUBw.COM 本站所有资源均来自网络，如有侵权，请联系删除。

本站所有资源均来自网络，如有侵权，请联系删除。

00A 000000

0000000000Redis0000000000000000000000300000003000
000000Redis000000000000Python00Python000Redis000000
0000000000000000000000000000000000

0000ePUBw.COM0000ePUBw.COM 0000

0000000000000000

A.1 Debian Linux Ubuntu Linux

Redis

Debian Linux apt-get install redis-server Redis Debian Ubuntu Redis Ubuntu 10.4 apt-get install redis-server 2010 3 Redis 1.2.6

Redis Redis Redis Redis Python Redis

A-1 make

A-1 Debian Linux

```
~$ sudo apt-get update
~$ sudo apt-get install make gcc python-dev
```

❶ <http://redis.io/download> 下载最新版的 Redis。

❷ 编译 Redis。

❸ 安装 Python 的 Redis 模块。

❹ A-2 启动 Redis 服务器。

A-2 在 Linux 上安装 Redis

编译 Redis。

安装 Redis。

启动 Redis 服务器。

```
~:$ wget -q http://redis.googlecode.com/files/redis-2.6.9.tar.gz
~:$ tar -xzf redis-2.6.9.tar.gz
~:$ cd redis-2.6.9/
~/redis-2.6.9:$ make
cd src && make all
[trimmed]
make[1]: Leaving directory `~/redis-2.6.9/src'
~/redis-2.6.9:$ sudo make install
cd src && make install
[trimmed]
make[1]: Leaving directory `~/redis-2.6.9/src'
~/redis-2.6.9:$ redis-server redis.conf
[13792] 2 Feb 17:53:16.523 * Max number of open files set to 10032
[trimmed]
[13792] 2 Feb 17:53:16.529 * The server is now ready to accept
connections on port 6379
```

从 <http://redis.io/download> 下载最新版的 Redis。
本书写作时 Redis 的最新版本为 2.6。

解压源码。

注意观察编译消息，
这里不应该看到错误。

注意观察安装消息，这
里不应该看到错误。

通过日志确认 Redis
已经顺利启动。

❹ A-3 安装 Python 的 Redis 模块。

Ubuntu/Debian 安装 Python 2.6/2.7 的 Redis 模块。

Python 2.6/2.7 安装 Redis 模块。

setuptools 简单 helper package ① A-3 安装

Python 的 Redis 模块。

附录A-3 Linux Python Redis

通过运行 ez_setup 模块来下载并安装 setuptools。

redis 包为 Python 提供了一个连接至 Redis 的接口。

```
~:~$ wget -q http://peak.telecommunity.com/dist/ez_setup.py
~:~$ sudo python ez_setup.py
Downloading http://pypi.python.org/packages/2.7/s/setuptools/...
[trimmed]
Finished processing dependencies for setuptools==0.6c11
~:~$ sudo python -m easy_install redis hiredis
Searching for redis
[trimmed]
Finished processing dependencies for redis

Searching for hiredis
[trimmed]
Finished processing dependencies for hiredis
~:~$
```

下载 ez_setup 模块。

通过运行 setuptools 的 easy_install 包来安装 redis 包以及 hiredis 包。

hiredis 包是一个 C 库，它可以提高 Python 的 Redis 客户端库的速度。

Python Redis A.4 Python Redis Redis

ePUBw.COM ePUBw.COM

A.2 OS X Redis

Redis Python Redis

1 OS X Redis

2 Python Redis

Linux Redis OS X Xcode Xcode Linux 10 Xcode Redis

Redis OS X Rudix

Rudix Redis OS X A-4 Rudix Redis

A-4 OS X Redis

		下载用于安装 Rudix 的 引导脚本。
命令 Rudix 安装自身。	<pre>~:~\$ curl -O http://rudix.googlecode.com/hg/Ports/rudix/rudix.py [trimmed] ~:~\$ sudo python rudix.py install rudix Downloading rudix.googlecode.com/files/rudix-12.10-0.pkg [trimmed] installer: The install was successful. All done</pre>	Rudix 下载并 安装它自身。
命令 Rudix 安装 Redis。	<pre>~:~\$ sudo rudix install redis Downloading rudix.googlecode.com/files/redis-2.6.9-0.pkg [trimmed] installer: The install was successful. All done</pre>	Rudix 下载并 安装 Redis。
启动 Redis 服务器。	<pre>~:~\$ redis-server [699] 6 Feb 21:18:09 # Warning: no config file specified, using the default config. In order to specify a config file use 'redis-server /path/to/redis.conf' [699] 6 Feb 21:18:09 * Server started, Redis version 2.6.9 Δ [699] 6 Feb 21:18:09 * The server is now ready to accept connections on port 6379 [699] 6 Feb 21:18:09 - 0 clients connected (0 slaves), 922304 bytes in use</pre>	
Redis 使用默认配置启动并运行。		

安装 Redis 使用 Python 安装 Redis 10.6
 10.7 OS X Python 2.6 Python 2.7
 Python Redis command + T
 A-5 Redis

A-5 OS X Python Redis

通过 Rudix 安装名为 pip 的 Python 包管理器。	<pre>~:~\$ sudo rudix install pip Downloading rudix.googlecode.com/files/pip-1.1-1.pkg [trimmed] installer: The install was successful. All done</pre>	Rudix 正在 安装 pip。
	<pre>~:~\$ sudo pip install redis Downloading/unpacking redis [trimmed] Cleaning up... ~:~\$</pre>	现在可以使用 pip 来 为 Python 安装 Redis 客户端库了。
	Pip 正在为 Python 安 装 Redis 客户端库。	

Linux Windows Redis
 setuptools easy_install Redis

pip 安装 Redis 的 pip 包，然后使用 setuptools 安装 Redis
pip 安装 Redis 的 pip 包，然后使用 Python 安装 Redis，然后使用 setuptools
安装 Redis

Linux 安装 Redis 的步骤如下：Linux
安装 Redis 的步骤如下：hiredis 安装 Redis 的步骤如下：OS X 安装 Redis
的步骤如下：Redis 的步骤如下：—— 安装 Redis 的步骤如下：Xcode
安装 Redis 的步骤如下：

Python 安装 Redis 的步骤如下：A.4 安装 Redis
Python 安装 Redis 的步骤如下：

ePUBw.COM ePUBw.COM 安装 Redis

安装 Redis 的步骤如下：

A.3 Windows Redis

Windows Redis 安装和配置指南

- Windows Redis 安装
- Windows Redis 配置
- Windows Redis Python 接口
- Windows Redis 性能优化

Windows Redis 安装和配置指南

A.3.1 Windows Redis

Redis 安装和配置指南

Windows Redis 安装和配置指南

作者Dusan Majkic Redis Redis 2.4.5
Redis

Windows Redis Windows
Redis Redis Redis
<https://github.com/MSEOpenTech/redis/>
Visual Studio Visual Studio Express 2010
Redis Redis Redis

Windows Redis Windows
Redis

A.3.2 Windows Redis

作者Dusan Majkic GitHub
<https://github.com/dmajkic/redis/downloads>
Windows 32 64 Redis

zip Windows XP
Windows zip Windows XP
Windows zip 32 64
Redis redis-server Redis
64 Windows 32 64 Redis 32 Windows
32 Redis Redis A-1

安装Redis并安装Python



```
C:\redis\32bit\redis-server.exe
[936] 10 Jul 21:40:50 # Warning: no config file specified, using the default con
fig. In order to specify a config file use 'redis-server /path/to/redis.conf'
[936] 10 Jul 21:40:50 * Server started, Redis version 2.4.5
[936] 10 Jul 21:40:50 # Open data file dump.rdb: No such file or directory
[936] 10 Jul 21:40:50 * The server is now ready to accept connections on port 63
79
[936] 10 Jul 21:40:51 - 0 clients connected (0 slaves), 672768 bytes in use
```

A-1 Windows Redis

A.3.3 Windows Python

安装Python 2.6或Python 2.7并安装Python 2.7
安装Redis并安装Python
<http://www.python.org/download/> Windows 2.7
32位或64位安装程序.msi
安装

安装Python 2.7到C:\Python27\并安装
Python Redis并安装Python 2.6并安装
Python27并安装Python27并安装Python26

A.4 Redis

```
RedisPythonRedis
PythonA-7
PythonRedis
```

□□□□A-7 □□Python□□□Redis

启动 Python,并
使用它来验证
Redis 的各项功
能是否正常。

设置一个值,然后通过获取返回值来判断设置操作是否执行成功。

```

~:$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import redis
>>> conn = redis.Redis()
>>> conn.set('hello', 'world')
True
>>> conn.get('hello')
'world'

```

Python Python
Python Windows OS X Python Idle
Linux idle-python2.6 idle-python2.7
`python -m idlelib.idle` Idle
IPython Python

OS X **Windows** **Redis** **Windows** **OS X** **Redis**
2.4 **Redis** 2.6 **Redis**
Redis 2.4 **Redis** 2.6
3

Redis **Redis** **AOF**
SHUTDOWN **Redis**
Redis **redis.conf**
Redis 4

hiredis **Linux** **Windows** **OS X**
Debian/Ubuntu **Linux** **Python**
hiredis **C**
Windows **OS X**
Windows **OS X** **hiredis**

Python **Redis**
Python **Python**
Python **Python**
Python <http://docs.python.org/tutorial/modules.html>
6.1.1

Python 3.7 9.10 9.11

Redis Python 1

Redis Python Maning Redis in Action

① Python setuptools pip pip Python pip virtualenv

ePUBw.COM ePUBw.COM

[illegible]

11A Redis Redis

[illegible]

□□□□ePUBw.COM□□□□ePUBw.COM□□□□

□ □ □ □ □ □ □ □ □ □ □ □

B.1 资源资源资源

资源资源Redis资源资源资源资源资源资源资源资源资源资源

- <https://groups.google.com/forum/#!forum/redis-db>——Redis资源
- <http://www.manning-sandbox.com/forum.jspa?forumID=809>——Manning资源资源Redis资源Redis in Action资源资源

资源资源ePUBw.COM资源资源ePUBw.COM 资源资源

资源资源资源资源资源资源资源资源资源资源

B.2 资源

Redis 资源

- <http://redis.io/>——Redis 官网
- <http://redis.io/commands>——Redis 命令
- <http://redis.io/clients>——Redis 客户端
- <http://redis.io/documentation>——Redis 文档

Redis 社区

- <http://github.com/dmajkic/redis/>——Dusan Majkic 的 Redis 社区
- <http://github.com/MSSOpenTech/redis/>——Redis 社区

Python 资源

- <http://www.python.org/>——Python 官网
- <http://docs.python.org/>——Python 文档
- <http://docs.python.org/tutorial/>——Python 教程
- <http://docs.python.org/reference/>——Python 参考

- <http://mng.bz/TTKb>——[Python 3.6 快速上手](#)
- <http://mng.bz/l31v>——[Python 3.6 快速上手](#)
- <http://mng.bz/9wXM>——[Python 3.6 快速上手](#)
- <http://mng.bz/q7eo>——[Python 3.6 快速上手](#)
- <http://mng.bz/1jLF>——[Python 3.6 快速上手](#)
- <http://mng.bz/0rmB>——[List 列表 comprehension](#)
- <http://mng.bz/uldf>——[Python 3.6 快速上手](#)
- <http://mng.bz/1XMr>——[Python 3.6 快速上手](#)

[ePUBw.COM](#) [ePUBw.COM](#)

B.3 消息队列和消息代理

- <http://celeryproject.org/>——Python消息代理和Redis消息队列
- <https://github.com/josiahcarlson/rpqueue/>——轻量级Redis消息代理Python实现
- <https://github.com/resque/resque>——轻量级Ruby消息代理Redis实现
- <http://www.rabbitmq.com/>——消息代理和消息队列
- <http://activemq.apache.org/>——消息代理和消息队列
- <https://github.com/Doist/bitmapist>——轻量级Redis消息代理支持bitmap-enabled analytics

本站所有资源均来自于网络，如有侵权，请联系删除。

本站所有资源均来自于网络，如有侵权，请联系删除。

B.4 圖表繪圖庫

- <http://www.jqplot.com/>——基於jQuery的圖表繪圖庫
- <http://www.highcharts.com/>——基於HTML5的圖表繪圖庫
- <http://dygraphs.com/>——基於Dygraphs的圖表繪圖庫
- <http://d3js.org/>——基於D3.js的圖表繪圖庫
- <http://graphite.wikidot.com/>——基於Graphite的圖表繪圖庫

□□

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

B.5 网络

5个网络地址IP地址范围如下：
网络地址范围

- <http://dev.maxmind.com/geoip/geolite>——5个网络IP地址范围
- <http://www.hostip.info/dl/>——网络地址IP地址范围
- <http://software77.net/geo-ip/>——网络地址IP地址范围

网络地址 ePUBw.COM 网络地址 ePUBw.COM 网络地址

网络地址范围

B.6 Redis 项目列表

- <http://mng.bz/2ivv>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/LCgm>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/UgAD>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/1OJ7>—— Instagram 使用 Redis 构建的实时聊天应用
- <http://mng.bz/X564>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/oClc>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/07kX>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/4dgD>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/21iE>—— 使用 Redis 构建的实时聊天应用
- <http://mng.bz/L254>—— 使用 Redis 构建的实时聊天应用

□□□□ePUBw.COM□□□□ePUBw.COM □□□□

□□□□□□□□□□□□

111

□□□□□□□□□□□□□□□□contact@epubit.com.cn□□□□□□□□□□

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

□□□□□□□□□□□□□□□□□□□□ebook@epubit.com.cn□

[illegible]

- 1234@1234567
- QQ12368449889

☐ ☐ ☐ ☐ **ePUBw.COM** ☐ ☐ ☐ ☐ **ePUBw.COM** ☐ ☐ ☐ ☐

[illegible]